UNIVERSITY OF OSLO
Department of Informatics

Visualization of subsurface grids in Octave

Master thesis

Lars Jahr Røine

May 1, 2012

Abstract

This thesis investigates the topic of subsurface reservoir grid visualization. An open-source Matlab toolbox for simulating and visualizing reservoirs is introduced. To make this toolbox work in Octave, an open-source Matlab clone, it is necessary to improve Octave's visualization. This is done by implementing two new visualization functions, which can be used from Octave. The first method tries to offer the same functionality as Matlab, but better performance, by improving an Octave function. The goal of the second method is to offer more advanced visualization, by fully utilizing the graphics processing unit and modern rendering techniques. The methods are tested, and compared to Matlab and standard Octave. It is shown that the new methods accomplish what they aim to do, and it is concluded that by doing as much work as possible on the GPU, we get higher performance and open up for more advanced visualization of subsurface reservoirs.

Contents

List of Figures

List of Tables

Preface

This thesis is part of my master studies at the Department of Informatics at the University of Oslo. It has been developed at SINTEF ICT as part of the Matlab Reservoir Simulation Toolbox project [28]. It corresponds to twelve months of work over a period of one and a half years. All the work presented in the thesis has been carried out individually.

The topic of this thesis was determined after a meeting with my (then future) supervisors, Knut-Andreas Lie and Christopher Dyken, during the fall of 2010. I came to the meeting with a strong wish to learn more about visualization techniques, but without any concrete ideas for a project. They had a suggestion for a project which involved developing visualization methods for subsurface grids in Octave. I did not fully understand the topic, or scope, of the project when I accepted it, but I am happy to say that I have never regretted choosing this topic for my thesis. It has required hard work for a long period of time to finish this thesis, and even though the task at times has felt a bit overwhelming, I have always gone to bed looking forward to the next day.

Even though the work in this thesis has been carried out individually, I could never have finished the thesis without the support and help from many people. I would like to start by thanking my supervisors, Knut-Andreas Lie and Christopher Dyken, for always taking the time to offer guidance, giving suggestions, reading my work, and helping me with the problems I have encountered. I would also like to thank the rest of the employees at the Department of Applied Mathematics at SINTEF ICT for keeping an open door, and allowing me to bother them with my questions, especially Halvor Møll Nilsen, Bård Skaflestad and Erik W. Bjønnes. Special thanks goes to André R. Brodtkorb for taking the time to organize a course in GPU-programming and Jon Hjelmervik for help with the examination. SINTEF ICT also deserves a special thanks for providing me with equipment and office space.

A big thanks goes to all my fellow students and friends through the past five years. I would especially like to thank Fredrik H. Valdmanis and Anders E. Johansen for being great friends, study partners, and helping me read my drafts. Thanks also goes to my fellow master students at SINTEF, in particular Simen A. A. Lønsethagen, for interesting discussions and good social environment.

I wish to thank my beloved Kristina for great support, incredible kindness and impressive patience. You are a very important part of my life.

Finally, I want to thank my family. We have been through a lot, and without your continued support, love, encouragement and advice I would not be where I am today.

Chapter 1

Introduction

This thesis deals with the topic of visualization of subsurface grids. Reservoir simulation is a topical field of research, and the visualization of reservoir models, or grids, is central to this topic. There are a large number of commercial software products for performing reservoir simulation and visualization, but most of these are proprietary and closed-source products. Open-source products are however becoming more popular within geoscience [12], and in an academic setting it makes sense to develop methods that are free and open for others to modify and improve.

1.1 Research questions

This thesis uses as a starting point the *Matlab Reservoir Simulation Toolbox* (MRST), which is an open-source toolbox of Matlab routines for performing reservoir simulation and visualization. It is developed at the Department of Applied Mathematics at SINTEF [28]. There is however a slight contradiction in the therm "open-source Matlab toolbox", since Matlab is closed-source, non-free software. A possible way to avoid this contradiction, and make the toolbox not dependant on proprietary software, is to use an open-source Matlab "clone" in combination with MRST. These are programs that offer more or less the same functionality as Matlab, but are open source. One of the most widely used alternatives, and the alternative that probably offers the highest compatibility with Matlab, is GNU Octave [7]. Even though Octave is not completely compatible with Matlab, most of the routines in MRST can be run directly in Octave, and routines that do not run directly usually need only small modifications. One of the great challenges with using Octave however, is the visualization routines. The visualization routines in MRST are already quite limited by Matlab, which does not offer high performance visualization or advanced volume visualization, but they are even more limited when using Octave.

For Octave to become a real alternative to use together with MRST, the

visualization will need to be improved. Furthermore, Octave would be an even stronger candidate if it provided more advanced visualization functionality than what can be provided by Matlab. The thesis tries to answer the following questions:

1. Is it possible develop new methods for visualizing MRST grid properties and simulation results in Octave that makes Octave's visualization functionality equal to that of Matlab?

2. Are we able to develop new methods that support more advanced visualization by using current hardware and modern rendering techniques, and are we able to integrate these methods into Octave?

These questions both have two parts. The focus of the first part of the questions is based on the field of visualization and computational geometry, where we are interested in developing the actual methods. The second part deals with the problem of integrating the new methods in Octave.

In newer versions of Octave the visualization is performed by using OpenGL [20], a cross-platform API for crating 2D and 3D graphics applications. The OpenGL version used in Octave, however, is old and does not make full use of the power in today's GPUs. Our hypothesis is therefore that by using functionality introduced in newer versions of OpenGL and modern rendering techniques, we will be able to improve the visualization in Octave. There are several advanced visualization features that would be relevant to introduce for MRST that are not supported in MATLAB. These include interactive cut planes, isosurfaces, and volume visualization.

GNU Octave is a large project, and we do not initially know how difficult it will be to modify its visualization system with our own methods.

1.2 Organization of thesis

This thesis is organized in eight chapters and divided into three main parts: background information, implementation, and results and discussion. In the first part there are four chapters that introduce the relevant background information we need to answer the research questions. In Chapter 2, we give a short introduction to reservoir simulation to explain why the topic of the thesis is relevant. Chapter 3 introduces the Matlab Reservoir Simulation Toolbox with special focus on the representation of grids. GNU Octave, and especially its visualization code, is covered in Chapter 4. Chapter 5 discusses OpenGL and the change it has gone through in recent years, which will help us understand the performance limitations on visualization code in Octave and how this can be improved.

The second part of the thesis describes the development and implementation of methods that have been made to try to answer the research questions.

The third part consists of two chapters. In Chapter 7, we describe tests that have been conducted to compare the functionality and performance of the various methods, and the results from these tests. Finally, the findings of the thesis and the answers to the research questions are summarized in Chapter 8.

Part I

Background

Chapter 2

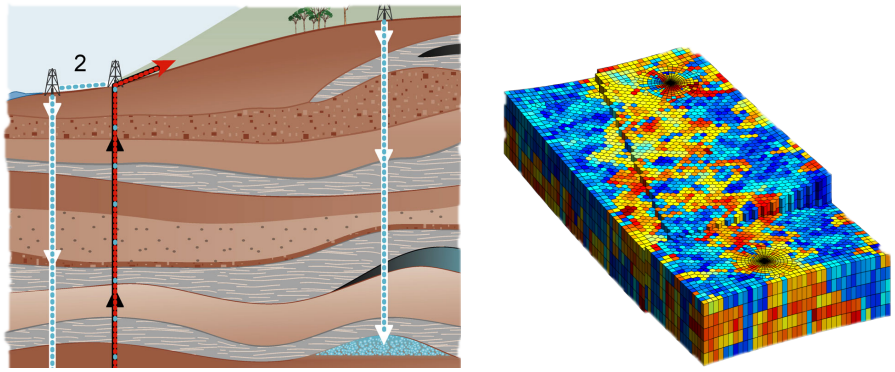
Reservoir Simulation

This chapter introduces the topic of reservoir simulation with special focus on how a simulation model arises. We also will also briefly discuss already existing software packages for performing reservoir simulation and visualization.

2.1 Simulation model and reservoir simulation

Reservoir simulation is an important field of study in today's society for several reasons. Our modern world is heavily dependent on various sources of energy, and petroleum is, and will probably continue to be, one of our most important energy sources. Reservoir modeling is important for efficiently extracting as much petroleum as possible from existing reservoirs. Another important application of reservoir modeling and simulation is research concerned with CO_2 storage in geological formations. By exploiting the capacity of subsurface reservoirs for storing CO_2 we might be able to avoid large-scale climate changes while moving out of the petroleum era [4, p.5]. Regardless of whether our aim is to extract petroleum or to store CO_2, reservoir modeling is heavily dependent on numerical calculations. The core of these calculations is to solve various partial differential equations, mainly to determine the fluid flow in porous media. Modern computers allow us to solve increasingly complex problems with ever increasing accuracy, however, no matter how powerful our computational tools are, they are of little use if we are unable to interpret the results. One of the most central tools we have for understanding the results from numerical calculations is visualization. With a good visualization tool, we are able to transform millions of digits into something easily understandable. Before we start discussing the details of reservoir visualization, we will have a closer look at how the model of the reservoir arises.

We start by looking at how the geological model is developed. When creating a geological model for a reservoir, we gather information by using various



(a) Cross section (b) Permeability plot

Figure 2.1: The figure on the left hand side illustrates a cross section of various layers. Seismic surveys give information of horizons and faults. Based on this information a grid can be built. Then, the subsurface grid is populated with the petrophysical properties. A permeability plot of a grid is shown in the figure on the right hand side.

methods. Seismic surveys provide useful information, but they have limited resolution, and are quite costly [15, p. 13]. Samples taken from already existing wells give very valuable information down to a finer scale than what seismic surveys give, and cores taken from the wells give even more detailed information. Geologists study geological analogues found elsewhere and use, among other things, information of the depositional environment. All this information is combined with the seismic surveys, and based on the information a geologist can create a rough model. From this rough model it is possible to create a subsurface grid. By combining the results from seismic surveys and the information from well samples, geostatistical methods can be used to give the grid petrophysical properties. The reservoir is then modelled as a volumetric grid with petrophysical properties, see Figure 2.1. The grid describes the structure of the reservoir and the mentioned petrophysical properties, such as porosity and permeability, are associated with the individual cells. This model, called a geological model, is a starting point for the simulation model.

Our geological model will typically represent an area covering several kilometers. The rocks a reservoir consists of are typically heterogeneous even down to the micrometer scale of pore channels. The grid cells in the simulation model will be on the meter scale. Thus it should be quite evident that some sort of upscaling is necessary to make use of the information that is found on the microscopic and mesoscopic models, which are illustrated to the left in Figure 2.2, in our simulation model.

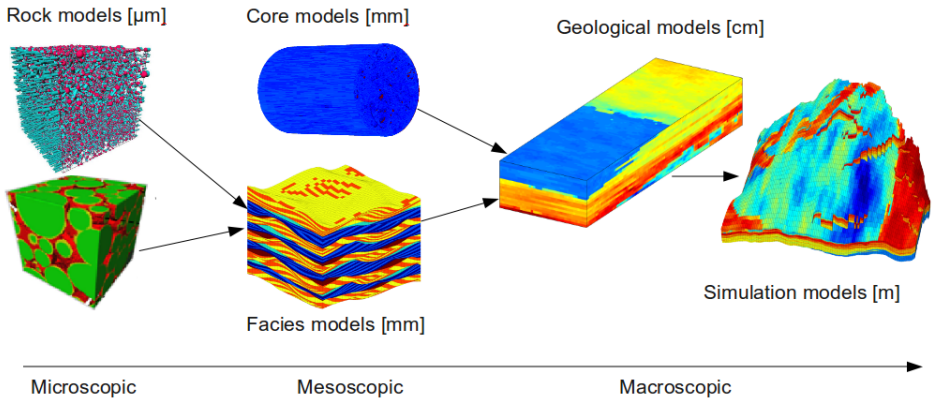


Figure 2.2: Upscaling. Image from Lie et al. [15]

We therefore need a way to upscale information, so that we can create a usable basis unit in our simulation model. This is done by exploiting the fact that rocks created at the same time will have similar properties. Though, because of movement in the continental planes and volcanic activity, rocks that were created at the same point in history will not necessarily be located next to each other. This means that, to find a way to represent the reservoir so that it can be used in simulation, it is necessary to find out how big we can make the representative elementary volumes (REV) of the simulation model without losing too much information. We do this by repeated upscaling of models on increasing scales as is illustrated in Figure 2.2. In the end, we arrive at a model that can be used for simulation which contains the needed information both with respect to properties of the rocks and geometry of the reservoir.

To perform simulations on a reservoir model and visualize the results, we are basically interested in describing the fluid flow and pressure of the reservoir. So far, our geological model only contains petrophysical properties, which are necessary, but not sufficient, to model fluid flow of the reservoir. We need a set of partial differential equations to model this fluid flow.

The void volume of the porous rocks of the reservoir is assumed to be filled with different phases. When we are modelling a reservoir we usually consider only three phases [1]: aqueous, oleic, and gaseous phase.

For each phase we use the continuity equation which says that mass is conserved

$$\frac{\partial(\phi\rho_i)}{\partial t} + \nabla \cdot (\rho_i v_i) = q.$$

Here, ϕ denotes porosity, ρ_i density, v_i flow velocity, and q models sources and sinks. To simplify matters, we choose to only look at single-phase flow. The idea behind modeling of multiphase flow is the same, but will naturally involve

more terms. In the case of single-phase flow we can use the following version of the empirical Darcy's law to model the flow velocity,

$$v = -\frac{\mathbf{K}}{\mu}(\nabla p + \rho g \nabla z),$$

where $\mathbf{K}$ is the permeability, μ the viscosity, g the gravitational constant, p the overall reservoir pressure, and z is the spatial coordinate in the vertical direction. We can combine the mass conservation equation with Darcy's law to get an elliptical pressure equation. For multiphase flow, we will in addition get a fluid-transport equation.

We use a *finite-volume method* to discretize the equations. Finite-volume methods have a more physical motivation than the classical finite-difference methods. They are derived by conservation of physical quantities over cell volumes. In this discretization the pressure will be associated with cells Ω_i whereas it is common to express the velocity as flux through side surfaces $\partial\Omega_i$.

The results can then be visualized, by giving the cells, or faces, of the grid model different color values based on solution data.

2.2 Existing reservoir simulation software

Subsurface reservoir simulation and visualization is very important in the oil and gas industry, and there exist a large number of commercial software programs for this purpose. They have in common that they claim to offer simple-to-use interfaces, with advanced functionality for modelling and visualizing reservoir models. Because such programs are closed-source and proprietary, we cannot use them in this thesis as a foundation for building new visualization methods. Instead, we will utilize the Matlab Reservoir Simulation Toolbox, which is an open-source toolbox that is built upon Matlab (which is proprietary) but also works well with Octave (which is an open-source Matlab clone).

Chapter 3

Matlab Reservoir Simulation Toolbox

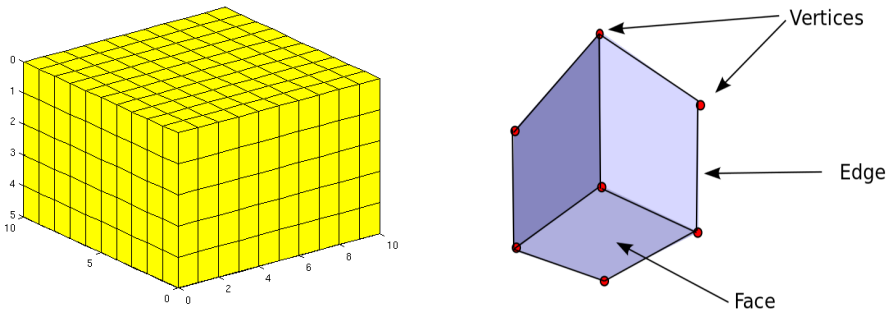
Having established some knowledge on how the geological and simulation model arise, we are ready to look at the software package which is the starting point of the thesis: The Matlab Reservoir Simulation Toolbox.

3.1 Introduction to MRST

The *Matlab Reservoir Simulation Toolbox* (MRST) is an open source toolbox containing Matlab routines and data structures for *reading, representing, processing, and visualizing unstructured grids, with particular emphasis on the corner-point format used within the petroleum industry* [28]. The toolbox is developed at the Department of Applied Mathematics at SINTEF ICT as a research (and teaching) tool for testing out ideas or creating prototypes. A large part of the MRST project is focused on various solvers. To be able to find new methods for visualizing sub-sea reservoir grids and to identify possible bottlenecks in the existing functions, we will need a good understanding of how the grids are represented in MRST. Our focus will therefore be on the underlying grid structure and the existing methods we have for visualizing the grids.

3.2 Grid types

A grid is a representation of a planar, or volumetric, object composed by cells. In 2D, these cells will be polygons, whereas in 3D, the cells will be closed polyhedra. The grid data structure in the MRST can be considered to be the most fundamental part of the toolbox [15, p.47]. Regardless of what kind of simulation, or visualization, we want to perform, we will need to pass a grid



(a) A regular Cartesian grid

(b) Elements of a cell

Figure 3.1: Grid cells.

instance to the various routines. We will now look at some types of grids MRST can represent.

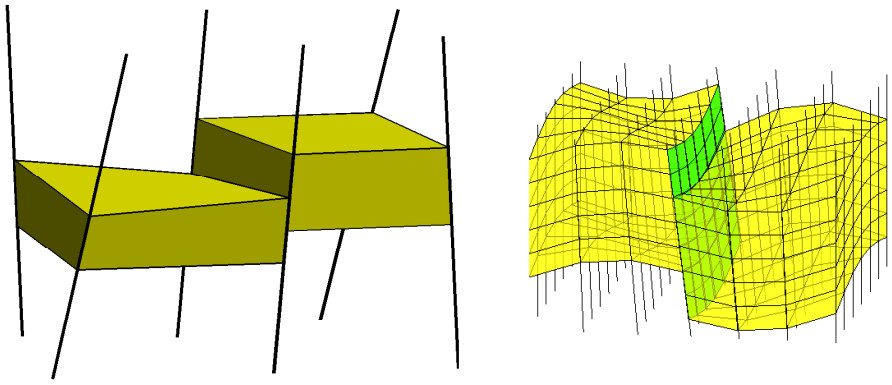
Structured grids

As the name suggests, a structured grid has a regular and repeating pattern, and consists of only one basic shape. The most basic form of a structured grid is a regular Cartesian grid, which in 3D consists of unit cubes. An example of a regular Cartesian grid can be seen in Figure 3.1a.

Other types of structured grids include rectilinear grids (tensor grids) and curvilinear grids. Rectilinear grids have nonuniform spacing between the vertices, in contrast to regular Cartesian grids which have uniform spacing between the vertices. Curvilinear grids have the same topological structure as regular Cartesian grids, but the cells are cuboids instead of parallelepipeds in 3D, and quadrilaterals instead of rectangles in 2D [15, page. 44]. The structured grids have in common that they can be referenced using an index tuple. This is easily seen by looking at a regular 3D Cartesian grid where the vertices have coordinates $(i\Delta x, j\Delta y, j\Delta z)$. When representing a grid of this type, it is only necessary to store the number of cells in x, y, and z direction together with the length of the cells in each direction. This would enable us to use a small amount of memory to store the grid, in addition to being an efficient method for accessing it. However, as we will see below, more complex grids demand a different representation. Therefore, in order to avoid several different grid representations, MRST stores all grids in a general unstructured format.

Unstructured grids

An unstructured grid is constructed from simple shapes that are laid out in an irregular pattern [15, p.47]. An unstructured grid in 3D will typically be made



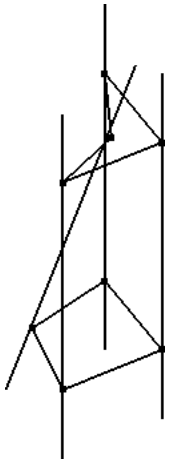
(a) Corner-point concept (b) Simple corner-point grid

Figure 3.2: To the left the corner point concept is illustrated. To the right a simple corner-point grid produced by MRST. The fault surface is marked in green.

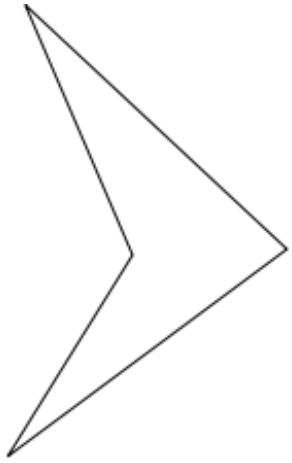
up of tetrahedrons. Structured grids are limited in what they can represent, and even though there are methods that can be used to model more complex geometries, unstructured grids are generally much more flexible. However, the strength of unstructured grids is also their drawback. By being more complex, they require a more complicated representation. In addition to information regarding cells, faces, and vertices, it is necessary to store information about the connectivity between these.

The industry standard for representing subsurface reservoirs geology is stratigraphic grids. These grids are built based on geological horizons and fault[1] surfaces [15, p.53]. An example of a stratigraphic grid is the corner-point grid, which is used for modeling the geological structures of petroleum reservoirs [22]. A corner-point grid is made out of a set of hexahedral cells and coordinate lines, as illustrated in Figure 3.2. The corners of the cells are attached to these lines (or pillars) and restricted by four pillars. Each cell will therefore have eight logical corner points. These corner points are called logical because in a corner-point grid the cells are allowed to collapse along the pillars to better model geological features like faults and layers. Although corner-point grids have a logical ijk-indexing, it is not sufficient to describe the grid connectivity if the reservoir model contains degeneracies like, for example, faults . We should here take note of the fact that faces of a corner-point grid cell might not be convex, as the pillars might be divergent. See Figure 3.3. A polygon is convex if all interior angles are less than 180 degrees, and non-convex (or concave) if at least one interior angle is greater than 180 degrees. This will be important to have in mind when rendering MRST grids.

[1] Fractures where the layers in the rock have been displaced [15]



(a) Corner point grid with diverged pillar



(c) Top face of corner point grid

Figure 3.3: Illustration of a corner-point grid cell with a divergent pillar. To the right we see the grid's top face which clearly is concave.

3.3 Grid representation in MRST

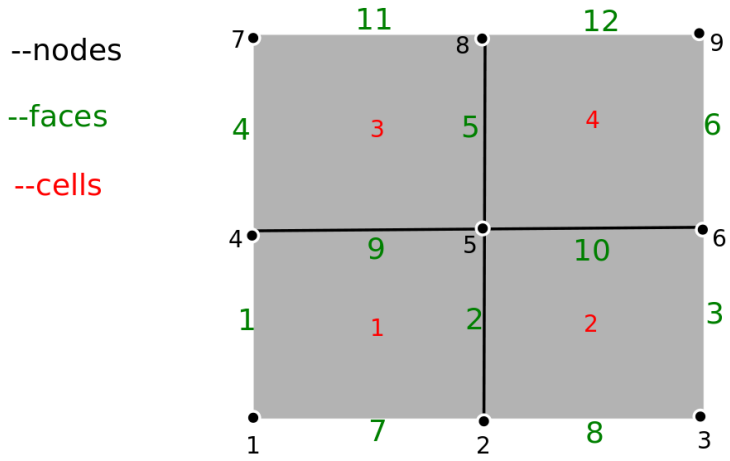
We now consider the grid data structure used by MRST to represent grids. The reason MRST chooses to store all grids as unstructured grids, even though the structured grids can be represented more effectively, is that the same solvers can be used, regardless of the underlying grid structure. To represent the necessary types of grids, both structured and unstructured, the grid data structure in MRST describes a set of connectivities. These connectivities describe which faces a cell consists of, which vertices form the different faces, and neighbor relations between the different elements. All this information is stored in a Matlab structure that has the main fields `cells`, `faces`, and `nodes`.

We will now have more a detailed look at how these various fields describe the grid by looking at an example. To avoid excessive indexing, we will look at a 2D Cartesian grid, but the principle is the same in three dimensions. We create the grid using the `cartGrid` command in MRST

```
g = cartGrid([2, 2])
```

The grid will have unit length on all faces. An illustration of the grid and an overview of the Matlab structure can be seen in Figure 3.4.

The Matlab structure we get when issuing the command above contains several variables. We will focus on the sub-structures `cells, faces`, and `nodes`, illustrated in Figure 3.5, as these are the most crucial for describing the geometry




```
g =

  cells: [1x1 struct]
  faces: [1x1 struct]
  nodes: [1x1 struct]
cartDims: [2 2]
   type: {'tensorGrid'  'cartGrid'}
griddim: 2
```

Figure 3.4: Illustration of a simple Cartesian grid with numbered cells, faces and nodes together with the Matlab structure that holds the grid information.

```
g.cells =

    num: 4
 facePos: [5x1 int32]
indexMap: [4x1 int32]
  faces: [16x2 int32]
```

```
g.faces =

     num: 12
  nodePos: [13x1 int32]
neighbors: [12x2 int32]
      tag: [12x1 int8]
    nodes: [24x1 int32]
```

```
g.nodes =

    num: 9
coords: [9x2 double]
```

Figure 3.5: Contents of the three structures cells, faces and nodes.

of the grid. These structures are related in an hierarchical manner. In order to find the coordinates of the nodes that make up a certain cell, we have to use all three structures. After having described the contents of each structure this will be exemplified.

The `cells` structure has four fields, one integer holding the number of cells, and three arrays. The array `indexMap` is only relevant if the grid has both active and inactive cells. Then this array will hold the indices of the active cells, sorted in ascending order. The two arrays `facePos` and `faces` are closely related. The `facePos` array is an indirection map into the `faces` array. This means that we use this array as an auxiliary structure when indexing the `faces` array to find which faces a cell consists of. We get the submatrix which gives us the face information about cell i by indexing the `faces` array in the following manner

```
faces(facePos(i):facePos(i+1)-1, :)
```

The `facePos` array is constructed in such a way that we can find the number of faces for each cell by running the Matlab/Octave command

```
diff(facePos)
```

The `faces` matrix is a $n \times 3$ matrix, where n is the number of faces. It gives information about which cell a given face is part of, and the direction of that face. The first column denotes which cell a face belongs to, the second column holds the global face number, and the third column, which is optional, describes the direction of the face. The first column is omitted due to memory reasons, as it can be reconstructed using the `facePos` array

```
rldecode(1:g.cells.num, diff(g.cells.facePos), 2).'
```

The matrix for the example grid, with the reconstructed first column, can be seen in Figure 3.6. Comparing the output above with the illustration of the grid in Figure 3.4, we see that the second column holds a global face number, and that the first column holds the global cell number which each face is connected to. For example, we see that global cell number 1 is made of of the global faces 1, 7, 2 and 9.

Moving on to the `faces` structure (not to be confused with the array faces in the `cells` structure) we find a layout similar to that in the `cells` structure. Here the arrays `nodePos` and `nodes` are connected in a similar manner as `facePos` and `faces`, with the difference that these arrays hold information about which nodes the different faces are made up from. The indexing of these arrays and the reconstruction of the first column in `nodes` is done in the same manner as explained above. One difference is that the `nodes` array only has two columns, as there is no direction information. In addition, we have the array `neighbors` that can be used to find the neighboring cells of a face (one or two). That is, `neighbors(i, :)` lets us know the global cell number face i is connected to. A zero in one of the columns means that the face is only connected to one cell. The `nodes` and `faces` arrays for Figure 3.4 can be seen below.

g.cells.faces =			g.cells.facePos
1	1	1	1
1	7	3	5
1	2	2	9
1	9	4	13
2	2	1	17
2	8	3	
2	3	2	
2	10	4	
3	4	1	
3	9	3	
3	5	2	
3	11	4	
4	5	1	
4	10	3	
4	6	2	
4	12	4	

Figure 3.6: Contents of the `faces` and `facePos` arrays of the cell structure.

The final of the three structures, `nodes`, contains only two fields: an integer with the total number of nodes in the grid and an array with the coordinates of the nodes.

By combining the fields in these structures we are able to extract information about the grids that might be of interest. We could, for instance, be interested in finding the coordinates of the nodes of a cell. We will use cell number two in Figure 3.4 as an example and find the coordinates of its nodes. We start by extracting all the faces the cell consists of. In this example we assume that the first column in the arrays `faces` and `nodes` has been generated.

```
face_info = g.cells.faces(...
g.cells.facePos(2):g.cells.facePos(2 + 1)-1, :);
```

This yields a matrix with the information regarding the relevant faces, see Figure 3.8. The global face number the cell consists of is found in the second column. Now that we have the faces, we can use each of them and find which nodes that make up the individual faces. To avoid too much output we choose one of the faces, as the procedure is identical for the other. As can be seen from Figure 3.4 and the `face_info` matrix in Figure 3.8, one of the faces in the `face_info` will be global face number two. We then use the following code to extract the node information for face number two.

```
node_info = g.faces.nodes(...
g.faces.nodePos(face_info(1, 2)):...
g.faces.nodePos(face_info(1, 2)+1)-1, :);
```

g.faces.nodes

1	1
1	4
2	2
2	5
3	3
3	6
4	4
4	7
5	5
5	8
6	6
6	9
7	2
7	1
8	3
8	2
9	5
9	4
10	6
10	5
11	8
11	7
12	9
12	8

g.faces.neighbors

0	1
1	2
2	0
0	3
3	4
4	0
0	1
0	2
1	3
2	4
3	0
4	0

Figure 3.7: Output of the matrices that hold node and neighbor information for the faces of a simple Cartesian grid.

From the `node_info` matrix in Figure 3.8 we see that face number two of cell number two is made up from nodes number 2 and 5. We can then use these numbers as indices in the `coord` array of the `node` structure to find the final coordinates. Since the faces in Figure 3.4 has unit length we see that we have arrived at the correct coordinates for nodes number two and five. It should be noted that additional fields can be added to the grid structure. Examples include face areas and cell volumes, which can be added by running the `computeGeometry` command. We will however not discuss these further as they are not necessary when representing the grid visually.

3.4 Grid visualization in MRST

There are several functions in MRST that can be used either separately or in combination to produce grid plots. The different functions focus on different parts of the grid visualization. The plot functions in MRST have a tree-based structure. An illustration of the plot functions and how they relate to each

```
face_info =                    node_info =          g.nodes.coords(2, :) =

   2          2          1        2          2          1     0
   2          8          3        2          5
   2          3          2                          g.nodes.coords(5, :) =
   2         10          4
                                                                 1     1
```

Figure 3.8: Finding coordinates to nodes of a cell.

other is shown in Figure 3.9. They all have in common that they take a grid structure as their primary argument and that they depend on the low-level Matlab function `patch` to carry out the actual plotting. The `patch` function creates a 2D or 3D "patch", which essentially is a filled polygon. If the grid is in 3D, which it usually will be, the polygon will be embedded in 3D space. It accepts data specified using the properties *Faces*, *Vertices* and *FaceVertexCData* (color information). *Vertices* is as the name suggests a matrix with all the vertices in the grid and *Faces* is an index list specifying which vertices are used to form the different faces. All the plotting functions eventually provide the data they want to visualize on this form. This means that the data to be rendered is based on the faces of the grid. Most of the functions accomplish this by calling `plotFaces` as an intermediate step. The `plotFaces` function extracts vertex and face information from the grid structure, together with color data, to form the correct arguments for a call to the `patch` function. A list of the global face numbers to be plotted can be provided as argument. However, it would be cumbersome to always specify arguments on a per face basis, for example if we want to plot only certain parts of the grid or give color information per cell. Therefore there are a set of plotting functions that are built on top of the `plotFaces` function that accept "higher level" input. As an example we consider the `plotCellData` function. This function makes it possible to provide per cell coloring and specify a subset of the grid to plot. To save resources this function, together with `plotFaceData` and `plotGrid`, extracts the boundary faces of the grid, as illustrated in Figure 3.10. The input data to this function is then reorganized to be per face and sent to the `plotFaces` function.

3.5 Limitations on the grid visualization in MRST

The MRST provides different visualization routines that can be combined to create different types of grid visualization. There are, however, some limitations that would have been beneficial to improve.

One of the first things one notices when rendering a grid of medium size, is that interaction with the grid, for example rotation, is difficult, due to a low frame rate. This makes it difficult to inspect the grid from a certain view-

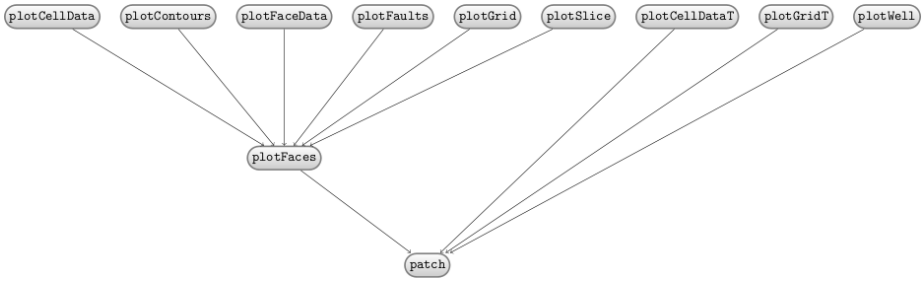
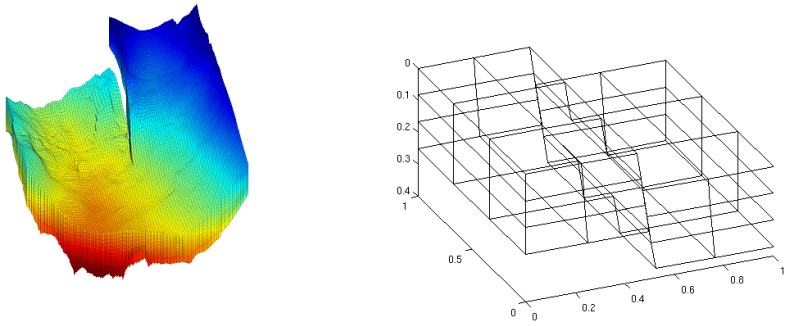


Figure 3.9: Hierarchy of plotting routines in MRST.



(a) The Johansen formation

(b) Boundary faces only

Figure 3.10: On the left we see a height plot of the Johansen formation made using the `plotCellData` function. To the right we see a plot of a simple grid produced by the `plotGrid` command. By studying the plot it is possible to see that only the exterior of the grid has been plotted.

point. To counter this problem, the performance of the plotting routines must be improved.

Another limitation is that the visualization of grids is script based, meaning that to change anything about the grid we are visualizing, we must give written commands. Sometimes it would have been more intuitive if it was possible to do this through the visualization tool by clicking on buttons, or using the mouse cursor.

Chapter 4

GNU Octave

Now that we have been introduced to the Matlab Reservoir Simulation Toolbox it is time to take a closer look at GNU Octave.

GNU Octave is a high level interpreted programming language directed at numerical computations. It was originally developed to be a companion software to a university course on chemical reactor design, but has later developed into being a popular language used in teaching, research and commercial applications [7]. GNU Octave is very similar to MATLAB in use, and the Octave developers consider any difference between the two as a bug. This high degree of compatibility makes Octave a possible alternative to Matlab for use together with MRST. The use of MRST in combination with Octave does, however, present some challenges. One of the biggest challenges of using Octave together with MRST is that the plotting functionality in Octave is poor. Large grids are rendered very slowly which makes it hard for the user to actively interact with the plots. In addition, Octave does not fully support modifying face and edge alpha values, which is often used when plotting grid properties.

In this chapter, we will investigate the reasons for choosing Octave, and lay the foundation for uncovering why the plotting performance is so poor. The structure of the visualization code will be presented, so that we have a foundation for discussing how to implement new visualization methods.

4.1 Why Octave?

Although the use of Octave together with MRST is not without certain challenges, using Octave instead of Matlab has several advantages. Octave is released under the GNU General Public License, which means that it can be freely distributed and modified when following this license. This makes Octave an interesting alternative for commercial companies since it will reduce the cost associated with buying licenses for commercial software. In the context of the

thesis, Octave is an interesting alternative to Matlab since it gives us greater flexibility when trying to implement new visualization methods. We have the opportunity to modify and add to the existing plotting functionality, we can integrate our own plotting backend, or we can use Octave's *Oct*-interface that allows us to write C++ code that can be dynamically linked with Octave. This will hopefully allow us to write new plotting routines using C++ and OpenGL and make them a completely integrated part of Octave, something we cannot fully achieve with Matlab. In addition, MRST is an open source toolbox, and following this philosophy it seems only natural that it should be used by an open-source programming language like Octave.

4.2 MRST and Rendering in Octave

Octave currently has two built in alternatives for plotting: Gnuplot [11] and a FLTK/OpenGL based plotting system. The plotting system based on FLTK (Fast Light Toolkit) [8] and OpenGL was first introduced in Octave 3.2.

As we discussed in Section 3.4, all MRST plot functions are based on Matlab's `patch` function. This makes the use of Gnuplot as a plotting system in combination with MRST impossible: Gnuplot supports only triangular patches in 3D, but MRST requires polygonal patches with more than three sides. The FLTK/OpenGL alternative, however, supports the same use of the `patch` function as Matlab and makes it possible to use MRST plotting commands in Octave. As we will see from the results in Chapter 7 there are however good reasons for implementing new plotting functionality in Octave, since the existing implementation of `patch` offers poor performance. This can either be done by implementing new plotting functionality side by side with the current OpenGL code, thus using FLTK, by creating a new backend, or using the oct-interface to communicate with a standalone application. If we use the FLTK/OpenGL plotting system as a starting point when adding our own customized plotting functions it will be seamlessly integrated into Octave. As an effect it will be possible to display plots using existing and new plotting functionality in the same window. This makes it possible to use the new plotting functionality to display the grid, but existing functionality to plot, for example, wells. The disadvantage of extending already existing code is that we are somewhat restricted to a large framework that might not be well suited to the functionality we wish to implement. We also run the risk of changing too much of the Octave source code, so that it will be difficult to transfer the code to newer versions of Octave. In addition, there might be a performance penalty compared to using a standalone application tailor-made for MRST plotting routines. This will be investigated in Chapter 7.

4.3 Octave's internal structure

If we want to implement new plotting functionality inside the already existing Octave code base, it is necessary to have an understanding of the inner workings of Octave. Octave is a fairly large project written mainly in C++. It consists of C++ files in addition to a large number of dynamically linked libraries and script files written in Octave. The source code is available from the Octave project's homepage [7]. The organization of the source code follows a fairly standard layout and uses GNU Automake [10] and GNU Autoconf [9]. The core of Octave is contained in the `src` folder, where we find the interpreter, a lot of infrastructure needed by the interpreter, Octave's own type system and the built-in functions. It is the latter that is the most interesting for us since the built-in functions include the graphics toolkit and the render code. A complete discussion of the organization and functionality of Octave's source code is beyond the scope of this thesis. We will however need to take a look at the structure of the plotting code, as this is what we want to improve.

4.3.1 Structure of Octave's plotting system

Since Octave inherently supports two different graphics toolkits, it is no surprise that the plotting code is organized in such a way that as much code as possible is shared by the two toolkits. The shared plotting code is built around *graphics objects* and handles to these graphics objects. The handles and graphics objects are stored in a C++ STL[1] map which can by used by both graphics toolkits.

We will now look at the most central steps that are performed when the following commands are issued in the Octave interpreter, which draws a quadratic polygon using the `patch` command.

```
graphics_toolkit('fltk')
v = [0 0 0; 1 0 0; 1 1 0; 0 1 0];
f = [1 2 3 4];
patch('Vertices', v, 'Faces', f)
```

The `patch` command is an Octave script file that checks that the input to the function is valid and initializes the correct graphics toolkit, in this case FLTK. Then the type of graphics object that we want to make (a patch object), together with the arguments, are sent to a C++ function that starts the creation of a graphics object and a handle to that object. The complete patch graphics object will contain a variable of type *properties*. This member variable contains all the geometric data, and is what the render code will later use. After the graphics objects and handles are stored, the graphics toolkit takes control. It is to us only the FLTK toolkit that is relevant. The toolkit handles the window creation and user input. The FLTK toolkit has an object of type `opengl_renderer` which is where all the OpenGL render code is located. When it is time to

[1] Standard Template Library

draw the patch object the toolkit passes the graphics object as argument to the `opengl_renderer`'s `draw` function.

C++ code

```
void
opengl_renderer::draw (const graphics_object& go, bool
    toplevel);
```

The type of graphics object is determined, and a draw function that corresponds to the object's type is called, which in our example is the `draw_patch` function:

C++ code

```
  void
  opengl_renderer::draw_patch (const patch::properties
      &props)
```

It is worth noticing that this `draw_patch` function only has access to the properties variable of the graphics object, and that this variable is `const`, meaning that the contents cannot be changed from this function. In Chapter 6 we will discuss how to integrate our own rendering code into Octave's already existing plotting system.

4.4 Octave's C++ interface: oct-files

As an alternative to extending Octave's already existing plotting methods, we can extend Octave with a new visualization backend with new render methods. This would, however, require quite a bit of work, and the making of such a backend is outside the scope of the thesis. As an alternative we can use one of Octave's alternatives for including compiled code as a dynamical extension. Octave offers a mex-interface for integrating C-code very similar to that of Matlab. In addition, it is also possible to call C++ function through Octave's oct-file interface. This will allow us to send data from the Octave interpreter to a C++ application that can set up a simple window environment and use OpenGL for rendering. Although this solution will have some limitations compared to a complete graphics backend, it will illustrate what we can achieve when not being constrained by the Octave framework.

4.4.1 Inner workings of oct-files

To dynamically link a C++ function to Octave an oct-file must be generated. This is done by using the `mkoctfile` tool on a C++ file. This C++ file must include the macro `DEFUN_DLD` which defines the entry point into the dynamically loaded function [6]. As an example, consider this C++ file which defines a function that multiplies two square matrices and returns the result.

C++ code

```
#include <oct.h>
```

```
DEFUN_DLD(multiply_matrices, args, ,
          ``Add two matrices and return the resulting
              matrix.'')
{
  Matrix a = args(0).matrix_value();
  Matrix b = args(0).matrix_value();

  Matrix c = a*b;

  return octave_value(c);
}
```

The first argument to the `DEFUN_DLD`-macro is the name of the function, and must match the filename. The next argument is the input arguments from Octave. The third argument is optional and gives the number of output arguments. The last argument is a help-string which can by displayed in the Octave interpreter by typing `help function_name`. This C++ file can now be compiled with the `mkoctfile`-script which can be used inside the interpreter or from the shell, where it can be included, in for example, a Makefile. This will produce an oct-file, and as long as this file is in the Octave interpreter's path, it can be called as a regular octave function,

```
octave:1> A = [1 2; 3 4];
octave:2> B = [2 2; 2 2];
octave:3> C = multiply_matrices(A, B)
C =

    7   10
   15   22
```

Now that we have introduced Octave and the structure of its plotting system we are almost ready to look at how Octave implements the `patch` function, and develop possible improvements. First, however, we will have closer look at OpenGL and how it has evolved in later years.

Chapter 5

OpenGL

Both Octave and the new visualization methods described in Chapter 6 use OpenGL to render geometry. To understand the challenging aspects of Octave's plotting methods, and the reasons for making the decisions we do in Chapter 6, we need to understand what OpenGL is, and how it has changed in recent years.

OpenGL (Open Graphics Library) is a cross-platform API for producing 2D and 3D computer graphics. OpenGL provides access to the graphics hardware on a computer, making it possible to use various techniques to achieve very fast rendering of geometry. OpenGL has gone through a large amount of changes in recent years, the largest being the transformation from a fixed function pipeline to a programmable pipeline. Another big difference is that, starting from version 3.0, some features were marked as deprecated [20], meaning that they were expected to be removed in later versions. The deprecated functions are typically unused functions and functions that slows down performance. This has lead to the rise of two different profiles; *the compatibility profile* and *the core profile* [13]. Starting from OpenGL 3.2, the compatibility profile was introduced to give users access to functionality that has been marked deprecated, thus creating a *backwards-compatible profile* [26]. However, the producers of graphics drivers are not required to continue supporting deprecated functionality in the future, which means that the user is not guaranteed that all deprecated functionality will exist. The core profile is a more stripped down version of OpenGL, where all the deprecated functionality that does not abide modern GPU design has been removed.

In this chapter we will give a short introduction to the graphics pipeline and the OpenGL library. We also need to have a look at the OpenGL Shading Language (GLSL). We will also outline some reasons to why we can expect faster rendering when using some of the newer features of OpenGL. This knowledge will be important for understanding the choices we have made when im-

plementing the different methods, and when looking at reasons for improved performance. At the end of the chapter, we will look at some techniques for rendering polygons as this will be used in later chapters.

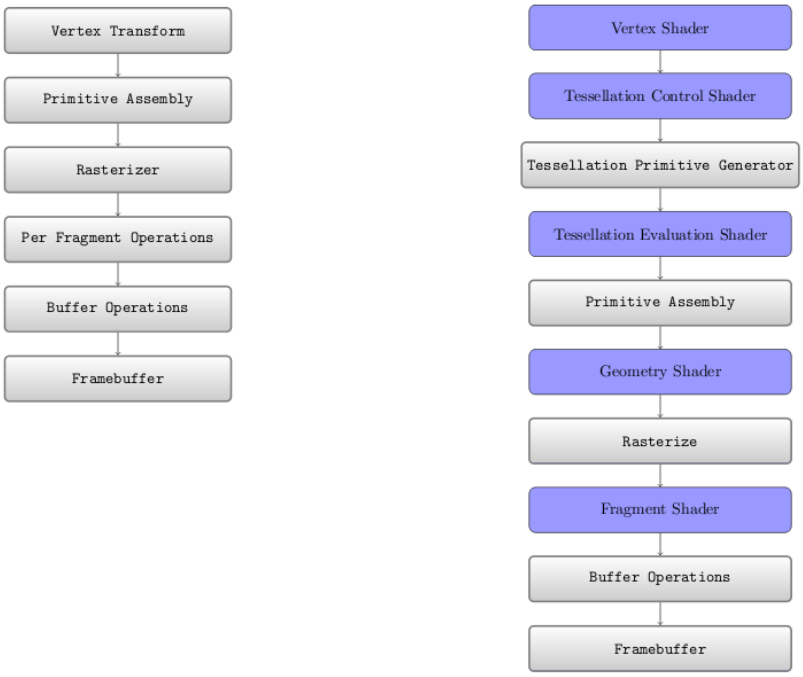
5.1 The graphics pipeline

The term *graphics pipeline* is a widely used abstraction to describe the inner workings of real-time graphics. The graphics pipeline describes, on a high level, what happens to the input data before it is rendered on screen. The graphics pipeline can be implemented in software or on the graphics processing unit (GPU). Originally, this pipeline description was closely related to how the hardware used to look like [17]. However, as the graphics hardware has evolved and moved towards a more unified design, we can no longer relate the graphics pipeline as closely to the hardware as before. Nevertheless, the graphics pipeline is still useful as an abstraction to understand how the geometry data is processed by the GPU.

5.1.1 The original graphics pipeline

The input data in OpenGL is a low-level description of the geometry (points, lines and triangles) and is sent from the application-controlled memory. This memory might be on the central processing unit (CPU) or in newer versions of OpenGL, on the GPU [24]. There are a number of ways to send geometry data to OpenGL, and some of these will be discussed later. Regardless of how the geometry data is sent, after OpenGL gets control of the data it is sent down the graphics pipeline, of which a simplified illustration is given in Figure 5.1a.

The first stage processes the vertex data. The vertices are transformed by several matrices that map the object-space coordinate system to the screen. Several additional vertex attributes are also set and modified, like color, normal vectors, and texture coordinates. After the vertices have been processed they are sent to the primitive assembly stage. Here, vertices are joined to form complete primitives (points, lines, triangles) before the primitives are *clipped* according to the view volume. Additional operations executed in these stages are perspective division and culling. The rasterize stage is where primitives are converted to a set of fragments. A fragment is a screen-space pixel position with interpolated vertex attributes. As an example consider a line which is described by two vertices that cover a certain number of pixels on the screen. The rasterization process produce fragments with these pixel positions [24]. After the rasterization stage, the fragments are sent to the per-fragment stage. One of the more important tasks in this stage is to determine the color of the fragment. This is often done by using interpolated data from previous stages or by using a *texture*-lookup. Using a texture can be thought of as attaching an image onto an object [2] and during the fragment processing it is decided which part of the image corresponds to each individual fragment.



(a) Fixed function pipeline

(b) Programmable pipeline, programmable stages marked with blue background

Figure 5.1: Simplified illustration of the graphics pipeline concept.

The framebuffer resides on GPU memory, and is where the values for every pixel is found. In the buffer operations stage, a number of tests are performed per fragment to decide if the fragment should be written to the framebuffer, and the fragment color can be blended with the framebuffer color. If a fragment passes the tests, it is written to the framebuffer.

5.1.2 The modern graphics pipeline

As mentioned above, the graphics hardware used to resemble the description of the pipeline in Figure 5.1a, with dedicated transistors per stage and a fixed data flow through the pipeline [17]. Developers had little or no possibility to customize the pipeline stages. This is no longer the case with today's hardware. Starting with NVIDIA's GeForce series, which was released in 2001 [25], developers had access to programmable vertex and pixel (fragment) shading stages. GPU architecture has moved towards a unified shader pipeline [16], and the same hardware is used to run the different stages of the pipeline. The term *shader* was introduced to describe a program that runs on the GPU and performs one of the stages in the graphics pipeline. Modern GPUs contain hundreds of cores that can execute in parallel [18], and shader programs run on these cores. By using shaders we can do more advanced computations on the GPU and benefit from its massively parallel computing power.

5.1.3 The stages of the modern pipeline

Figure 5.1b illustrates the current concept of the graphics pipeline. In recent versions of OpenGL, the programmer has gotten more control of the pipeline stages, and some new stages have been introduced. In the modern graphics pipeline the first stage is still a stage where the vertex data is processed, this is done by the vertex shader. However, now it is completely up to the user to send the vertex attributes and uniforms that are needed to this stage and multiply the vertices with the necessary transformation matrices.

The three next stages in Figure 5.1b were first introduced in OpenGL 4.0. These stages are not necessarily enabled, and when in use, they allow for hardware *tessellation* (see also Section 5.3). The tessellation shaders and the non-programmable tessellation primitive generator can for example be used to divide a surface into smaller sub surfaces, thus making the geometry more detailed. This means that tessellation stages can change the amount of geometry in the rendering pipeline, this is unique to the tessellation stages and the geometry shader.

The primitive assembly stage has the same responsibility in the modern graphics pipeline as it had in the old, namely to join vertices into complete primitives. After the primitives have been generated, they enter a new pipeline stage: The geometry shader. The geometry shader stage was introduced in OpenGL 3.2 [19] and is executed once per primitive. The geometry shader needs not output the

same primitive as it received as input and it can output zero, one or more primitives [30]. The primitives it outputs must however be of the same type. The geometry shader can thus be used to discard, or create additional geometry.

After the geometry shader has finished, the geometry is rasterized before the fragment shader executes. The fragment shader can perform the same operations as the per fragment operations in the fixed function pipeline, but it is now fully the programmers responsibility. The output from the fragment shader goes through the tests in the buffer operations stage before the value is written to the framebuffer.

5.1.4 The OpenGL Shading Language

Now that we have an understanding of the different stages in the graphics pipeline, we need to look at how we can create shader programs that perform the programmable stages of the pipeline. In combination with OpenGL, it is customary to use GLSL, the OpenGL Shading Language, which is part of the OpenGL standard [24]. GLSL is a collection of several closely related high level shading languages [14]. The different languages are used to write shaders for the different programmable pipeline stages and currently there are five different languages, one for each of the following programmable processors: Vertex, tessellation control, tessellation evaluation, geometry and fragment processors [14].

Language overview

GLSL has a C-like syntax, but differs from C in that it does not support pointers. It does, however, provide a number of data types very suited for graphics computations, including different matrix and vector types, and support for arithmetic operations on these. GLSL also has a number of built in variables it receives from the graphics pipeline. For example will a vertex shader know which vertex it processes from the variable `gl_VertexID`.

Data is sent from OpenGL to the shaders in one of three different ways: *attributes*, *uniforms* and *textures*. An attribute is data that changes per vertex [13]. An example of an attribute is the actual position of a vertex. A uniform is constant for a group of primitives. If the color does not change per vertex, but is the same for the entire geometry, it can be sent as a uniform. Another common usage of uniforms is transformation matrices in the vertex shader. The third way to make data available to a shader from OpenGL is by placing it in a texture. The most common usage for texture data is to place image data that is to be applied on the surface of a triangle by the fragment shader. The developer is however free to place other types of data in a texture, which can then be looked up in a shader, making textures quite versatile.

Compiling and linking GLSL source code

GLSL source code needs to be compiled and linked before it can be used. This is done by using OpenGL functions to send the shader source code to the graphics driver where it is compiled and linked [13]. Different types of shaders can be linked together to form a `program` that will be used to perform certain parts of the graphics pipeline.

5.2 OpenGL rendering

This section will deal with some of the general OpenGL techniques that are necessary to follow the implementation details in this thesis. It will also shed light on some of the differences between deprecated and more modern rendering techniques, and why the latter are preferable.

5.2.1 Immediate mode

From the discussion about the graphics pipeline above we know that vertices and primitives are central concepts. This is because in OpenGL, one must build the geometric model from a small set of primitives which are described by their vertices [27]. These primitives are shown in Figure 5.5. The standard way to provide these vertices has been to specify the type of primitive one wants to render, and supply the vertices one by one. To render a pyramid with triangular base one would then use the OpenGL commands shown in Figure 5.2. Here we notify OpenGL that we are rendering triangles, and provide the vertices of four triangles. When we write `glEnd()` we let OpenGL take control of the data. The data will be transferred to the GPU via the PCI Express Bus and go through the pipeline stages. This is known as immediate mode rendering. This code is fine for rendering small amounts of geometry, but when wanting to render thousands, or millions, of vertices this code has several drawbacks with regard to performance. This type of code will typically be situated in a render function, and will need to be called for each frame we render. That means that in a real-time application this code will be called constantly. This is a problem because the PCI Express Bus will become a bottleneck and hinder the performance of the GPU. To see why this is true we will consider the theoretical maximum speeds of how many triangles the PCI Express Bus 2.0 can transfer per second and how many triangles a modern graphics card like NVIDIA GeForce GTX 580 can render per second.

The GTX 580 graphics card can render two billion triangles per second [18]. We can represent a triangle of three floats in C++ using 36 bytes (three vertices of three floats). The transfer speed of the PCI Express 2.0 using 16 lanes (which is how much can be used with the GTX 580 graphics card) is 8 GB/s. As we can see from Table 5.1 the PCI Express Bus is more than eight times slower than the graphics card. This ratio gets worse when we introduce additional attributes

C++ code

```
glBegin(GL_TRIANGLES);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(4.0, 0.0, 0.0);
    glVertex3f(2.0, 3.5, 0.0);


    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(4.0, 0.0, 0.0);
    glVertex3f(2.0, 1.75, 3.5);


    glVertex3f(4.0, 0.0, 0.0);
    glVertex3f(2.0, 3.5, 0.0);
    glVertex3f(2.0, 1.75, 3.5);

    glVertex3f(2.0, 3.5, 0.0);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(2.0, 1.75, 3.5);

glEnd();
```

Figure 5.2: Immediate mode rendering of a pyramid with triangular base.

Table 5.1: Triangles per second for the PCI Express 2.0 and NVIDIA GeForce GTX 580.

PCI Express 2.0 (16 lanes)	238,805,970
NVIDIA GeForce GTX 580	2,000,000,000

like colors and normal vectors. To avoid the performance penalty imposed by immediate mode we store the geometry data on the GPU so that it only needs to be transferred once.

5.2.2 OpenGL Buffers

OpenGL buffers make it possible to store data directly in GPU memory. The data can then be moved around and used by the GPU without involving the CPU [13]. Buffers can be used to store many different kinds of data, including vertex, pixel and texture data. Instead of using immediate mode rendering, we can therefore transfer the data once to a buffer on GPU memory and use the same data for subsequent render calls. The price we pay for being more efficient is slightly more code than what is needed when using immediate mode. Before rendering the data we need to set up buffers and upload data to GPU memory, see Figure 5.3. Note that instead of duplicating the indices we introduce an index array to describe which vertices should be used to form a triangle. Once all the data has been transferred to GPU memory, we can render the triangle with a single call to the OpenGL function `glDrawElements` as illustrated in

Figure 5.4.

C++ code

```
// Handles to buffer objects
GLuint vertex_buffer;
GLuint vertex_element_buffer;

GLfloat vertices[] = {
      0.0, 0.0, 0.0
      4.0, 0.0, 0.0
      2.0, 3.5, 0.0
      2.0, 1.75, 3.5
    };

GLuint pyramid_indices = {
  0, 1, 2,
  0, 1, 3,
  1, 2, 3,
  2, 0, 3
};

// Generate buffer object names
glGenBuffers(1, &vertex_buffer);
glGenbuffers(1, &vertex_element_buffer);

// Bind the buffer to hold vertex data and upload to the
    GPU
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
    GL_STATIC_DRAW);

// Bind buffer to hold the pyramid's indices and upload to
     the GPU
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
    vertex_element_buffer);
glBufferData(GL_ELEMENT_ARRAU_BUFFER,
    sizeof(vertex_element_buffers),
              vertex_element_buffer, GL_STATIC_DRAW);
```

Figure 5.3: Setting up buffers and uploading data in preparation of rendering a pyramid.

5.3 Drawing polygons in OpenGL

When not using hardware tessellation, the following primitive types can be used in OpenGL draw commands: Points, lines, line strips, line loops, triangles, triangle strips, and triangle fans, see Figure 5.5. In this thesis the drawing of filled polygons is of special interest, since this is exactly what the faces of the grid cells are. From Figure 5.5, it is obvious that triangles can easily be drawn by OpenGL. The same is true for convex polygons, since they can be represented

C++ code

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
    vertex_element_buffer);
glDrawElements(GL_TRIANGLES, 12, GL_UNSINGED_INT, 0);
```

Figure 5.4: Render pyramid when vertex data is stored in GPU memory.

by a triangle fan. When dealing with non-convex polygons, however, we need to take extra care. Some simple concave polygons may be drawn by using a triangle strip, but we can easily find examples of concave polygons that cannot be drawn correctly as a triangle strip. See Figure 5.6 for an example. Therefore, if we want to draw an arbitrary polygon using OpenGL, we need to represent the polygon by OpenGL's primitive types. The solution is to *tessellate* our polygons. Tessellation is the process of splitting a surface into a set of polygons [2]. Since OpenGL is optimized for drawing triangles, we would like to *triangulate* our polygons, that is split the polygon into triangles. There exist several algorithms for triangulating polygons, and in Section 6.3.2 we will take a closer look at a specific algorithm. Since tessellation is a very common task to perform when using OpenGL, there are tools specifically designed to be used together with OpenGL. One of these is the `GLUtesselator`. The OpenGL Utility Library (GLU) is a graphics library built on top of OpenGL that provides higher level functionality than what is obtained by using OpenGL. One of the utilities provided by GLU is the `GLUtesselator` which tessellates arbitrary polygons so that they can be rendered by OpenGL. We will have a closer look at the `GLUtesselator` in Section 6.2.

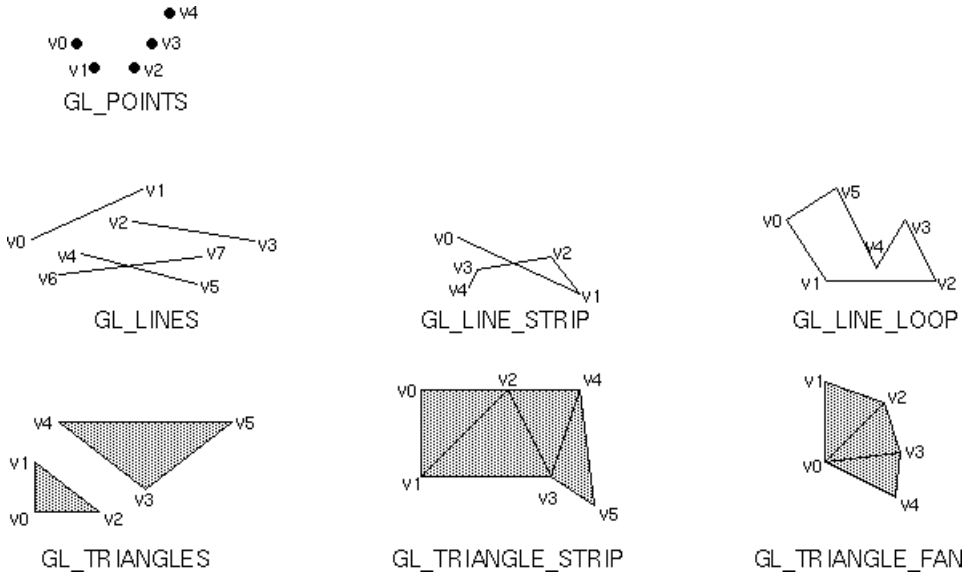


Figure 5.5: Geometric primitive types in OpenGL. The `GL_PATCHES` primitive which was introduced in OpenGL 4.0 is not depicted. Picture taken from The Red Book [27].

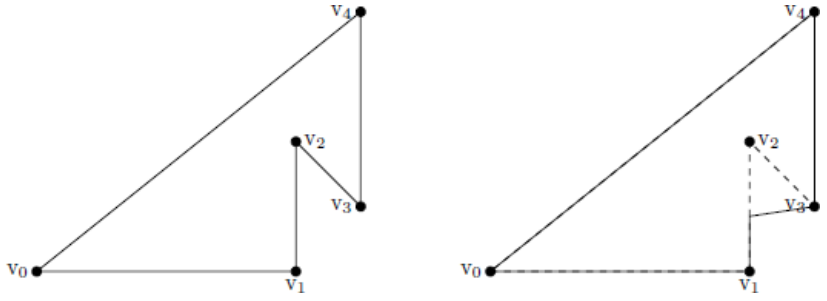


Figure 5.6: Illustration of what might happen when we use a triangle fan to represent a concave polygon. To the left we see a concave polygon. To the right the solid lines show how the polygon would look like if drawn using a triangle strip.

Part II

Implementation

Chapter 6

Implementation

In this chapter we will present the new methods that have been developed for visualizing MRST grids and how they are implemented in Octave. The programming language used is C++ and the new methods are based on OpenGL 4.2 and GLSL version 420. The methods have been integrated in version 3.6.1 of Octave. See Appendix A for instructions on how to obtain source code.

Before we start discussing the new methods we will take a closer look at Octave's implementation of the `patch` function. All the grid visualization routines in MRST are based on the `patch` function and by understanding the Octave implementation's weaknesses, we will be better suited to find improvements.

6.1 Implementation of `patch` in Octave

As we saw in Section 4.3.1, the contents of patch graphics objects is rendered in the `draw_patch` methods which is located in the `opengl_renderer` class. It has the following signature:

C++ code

```
void opengl_renderer::draw_patch (const patch::properties
    &props);
```

The input argument is an object which contains the information sent to the `patch` function from the Octave interpreter. The most important input, when called by MRST functions, is a Matrix (an Octave built in type) with vertices and a Matrix describing which vertices make up the individual faces. These faces are 3D polygons, and are rendered using OpenGL. As we saw in Chapter 5, OpenGL only accepts points, lines and triangles as input. The `draw_patch` function therefore uses `GLUtesselator` to *tessellate* the polygons before rendering them. We will have a closer look at how the `GLUtesselator` works in Section 6.3. The tessellation process is performed every time `draw_patch` is

called, regardless of whether it has been called before, which means that the geometry is tessellated every time the figure is rotated or the plot window is resized. In addition, the OpenGL code in `draw_patch` is in immediate, mode using `glBegin` and `glEnd`, which means that the data to be rendered is transferred to the GPU every time `draw_patch` runs. From Section 5.2.1, we know that in terms of performance, this is less than ideal. The biggest problem with the `patch` function in Octave is its performance. It offers a very low frame rate, which makes interaction with the model cumbersome. For the plot of the grid to be useful we will at least need to be able to rotate the model, so that we can inspect the different sides of the grid.

6.2 Improving the `patch` function

We have identified several problems with Octave's implementation of the `patch` function with respect to rendering performance. We will now present an alternative implementation of the `patch` function, specialized for rendering MRST grid properties. Then, after the method has been discussed, we will look at how we integrate this new method in Octave.

There are some quite obvious improvements that can be done on Octave's implementation: Use a tessellation algorithm only once instead of every time the figure is redrawn, and store the tessellated data on the GPU instead of transferring it every time the figure is drawn.

6.2.1 Tessellating and storing data

The alternative `patch` function will receive exactly the same input as Octave's built in `patch` function. The geometry is contained in the Octave Matrix types `vertices` and `faces`. The first thing we have to do is to tessellate the data, since we know from Section 3.2 that the cell faces of a MRST grid might be concave. However, instead of tessellating the data each time the `patch` function is called we will only tessellate the data when we receive it the first time. Octave is set up in such a way that the `patch` function is called each time the geometry needs to be redrawn, for example when resizing the window, zooming, or rotating the geometry. As long as the geometry does not change it is unnecessary to re-tessellate the data.

We use the same strategy as Octave to tessellate the polygons, namely the `GLUtesselator` which performs the tessellation automatically when given polygon data. The polygons that we want to tessellate (and draw) are contained in the matrices `vertices` and `faces`. Before sending the data to the `GLUtesselator` we create a `GLUtesselator` object and set up callback functions associated with this object. These callback functions are called at certain stages in the tessellation process and is how we decide what we want to do with the tessellated data. The three most central callback functions to set are `begin`, `end`, and

vertex, which are called at the start of a primitive, at the end of a primitive, and when a primitive's vertex is determined, respectively. These functions are called with different arguments depending on what the functions need. The GLUtesselator was created when immediate mode rendering was standard, so a (simplified) typical use of the tessellator would be the following,

C++ code

```
void begin(GLenum type)
{
  glBegin(type);
}

void end(void)
{
  glEnd();
}

void vertex (void *data)
{
  GLdouble vertex_data = (GLdouble) data;

  glVertex3dv(vertex_data);
}

void main()
{
  GLUtesselator* tess = gluNewTess();
  gluTessCallback(tess, GLU_TESS_BEGIN, begin);
  gluTessCallback(tess, GLU_TESS_END, end);
  gluTessCallback(tess, GLU_TESS_VERTEX, vertex);
}
```

This use of the GLUtesselator is conceptually similar to how Octave uses it in its implementation of the patch function.

To tessellate the geometry only once, we need to store the vertex data. This stored data can then be sent to the GPU and stored in a buffer. However, if we want to use a single OpenGL drawing command to render the data in the buffer, we must ensure that the tessellator does not produce a mix of, for example, triangles and triangle strips. This is something we do not have to think about when using immediate mode, since the callback associated with the start of a polygon receives a GLenum with the type of primitive the next vertices describe. We solve this problem by providing an extra callback function. If the tessellation object has a callback function associated with the type GLU_TESS_EDGE_FLAG it will only produce triangles. We therefore add such a callback function to the tessellation object.

Since we do not want to draw anything when we are given a new vertex by the tessellator, but rather store the vertex, we simply provide the tessellation object with a vertex-callback method that adds the vertices to a STL vector. When the callback functions are set up, we can input the polygons one by one

to the tessellator. However, we must take special care when doing this. The `GLUtesselator` requires that the vertices that represent the polygon must be in separate memory locations, because it does not copy the vertex data, only a pointer is saved. The representation of the faces given to the `patch` function is not in such a format. All the individual vertices are stored in the `vertex` Matrix, without any duplication, and the `faces` Matrix indicates which vertices to use for the different faces. The same vertex might therefore very well be used in different faces. Before we send the faces to the tessellator we must therefore combine the information found in `vertices` and `faces` to copy the vertices that form the different faces so that each face has its vertices in unique memory locations. When this is done, we loop over the faces and send the faces' vertices to the `GLUtesselator`.

After all the faces have been triangulated by the `GLUtesselator` we are left with a STL vector that can be transferred to the GPU and stored in a buffer and we can render it with a single OpenGL draw command. It is, however, not enough to only draw the faces of the grid. To be able to visually separate the faces from each other, or if we want to draw the grid with a wireframe representation, we also need to draw a line at the edges of the polygons. This cannot be done directly with the data we now have, since it is triangle-based. Drawing lines around all the triangles would not yield a correct visual result, see Figure 6.1. Therefore we need to tessellate the geometry one more time, but this time force the `GLUtesselator` to produce a line loop as primitive. This is done by setting a property of the tessellation object,

C++ code

```
gluTessProperty(tess, GLU_TESS_BOUNDARY_ONLY, GL_TRUE);
```

When the *boundary only* property is set to true, the tessellator will produce line loops that give us the outline of the polygon tessellated. We store the resulting vertices in a STL vector and transfer it to a buffer on the GPU. Since the number of boundary edges of a polygon will vary, we must also store the indices in the STL vector where a new line loop starts and the length of the line loop, as this is needed for rendering the lines correctly.

6.2.2 Implementing the alternative `patch` function in Octave

The new method we have developed must now be integrated into Octave. To make the management of the method easier we make a class, `patch_grid`, that manages geometric data, creation of buffers, compiling of shaders, and rendering. This class takes the same input as Octave's `patch` function, most importantly two Octave matrices with vertices and indices, together with any grid properties represented as color values. When an object of the class is initialized, the data is tessellated and transferred to buffers on the GPU. The data can then be rendered by calling the class' render method.

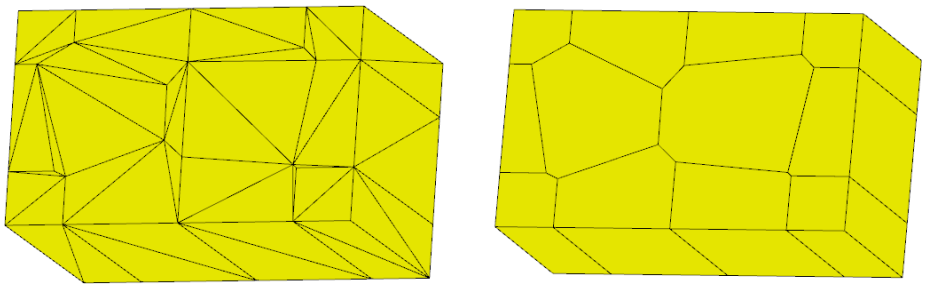


Figure 6.1: On the left we see the result we get when drawing all the triangles as lines. To the right we see the result when drawing only boundary edges of the triangles as lines

In Octave we make a new graphics object, which has a property object that holds all the information sent from the Octave interpreter when calling the new `patch2` function. We extend the `opengl_renderer` class, which performs all OpenGL rendering for the FLTK toolkit (see Section 4.3.1), with a new function

C++ code

```
void
opengl_renderer::draw_patch2 (const patch2::properties
    &props);
```

Every time a `patch2` object needs to be rendered this method is called. This program flow is customized for immediate mode rendering. Octave's `draw_patch` function would now tessellate the data, send it to the GPU and render. The key behind our new method, however, is that we only tessellate the data the first time this method is called. Subsequent times the function is called with the same patch2 graphics object, we only update the transformation matrices. Octave's way of organizing the data, by storing the geometry data sent from the Octave interpreter and processing it every time, does not fit the design of our new method. One alternative of implementing our new method that would fit it well in Octave's program flow would be to modify the `patch2::properties` class, so that the handles to GPU buffers and shader objects became a part of the input argument to the `draw_patch2` method. The first time the method was called with a new argument we could initialize the argument's `patch_grid` object, and subsequent times we could have called the render function of the `patch_grid` object contained in the input argument.

However, the input argument to the `draw_patch2` function must be prefixed with the `const` qualifier. The functions that the `draw_patch2` function depends on make it impossible to get rid of this qualifier unless we are prepared to rewrite large parts of Octave's internal structure. To break free of immediate mode rendering, we therefore need to violate Octave's practices on data flow. In the `opengl_renderer` class, which is not supposed to store geometry data,

we place a STL map with pointers to `patch_grid` objects as values. We equip each `patch2::properties` object with a unique integer handle, which we use as key in the STL map. Each time the `draw_patch2` method is called we check the argument's handle value. That way we can initialize a `patch_grid` object when we see it for the first time, otherwise we can retrieve the correct object and call its render method.

6.3 A new function for visualizing MRST grids

We believe that the alternative implementation of `patch` described above has the potential of making Octave's MRST visualization equal to that of Matlab. We will perform tests in Chapter 7, and this will help us answer the first of the research questions posed in Section 1.1. Now, however, we wish to enable more advanced visualization of MRST grids. To do that, we need to give the method access to the entire grid structure, not just the vertices and faces. By giving the method access to the entire grid structure we will hopefully be able to perform more of the processing of the grid on the GPU, and use the increased amount of information about the grid to offer more advanced visualization. Central ideas behind extraction of the boundary of the grid and the rendering of lines have been inspired by the Dyken [5].

The new method will be referred to as `plot_grid`, and will take an entire MRST grid struct as argument. We will then no longer be limited to working with only the parts of the grid sent to us from the Octave interpreter. The entire grid can then be tessellated and transferred to the GPU once. As example of more advanced usage we will introduce the option of interactively selecting a subset of the grid based on a plane equation. By keeping the whole grid on the GPU we can then render different parts of the grid without transferring more data to GPU memory. In contrast, to achieve the same effect when using one of the `patch` functions we would need to pick out the relevant cells in the Octave interpreter, and send the data to the `patch` function. Every time we change the plane equation, the data sent from the Octave interpreter would have to be tessellated and transferred to GPU memory.

6.3.1 Triangulating the grid

When we improved the `patch` function, the input we got was vertices and an index list specifying which vertices formed the different faces. Even though the input now is the entire grid structure, and we are able to perform more operations on the grid directly, we will still base the rendering on drawing the faces of each cell of the grid. Thus we still face the same challenges as we did when writing the alternative `patch` function: The faces must be triangulated before they can be rendered by OpenGL. The GLU Tesselator we used when implementing the alternative `patch` function is designed to handle not only concave polygons, but also polygons with holes and self-intersecting polygons [27, p. 505]. Since

the polygons in MRST grids will not self-intersect or contain holes, it should be safe to assume that by using a more specialized triangulation algorithm we can improve its performance. In addition, the GLU Tesselator outputs the new triangles by giving their vertices. We would rather have the indices of vertices as output, so that we do not have to store the same vertex several times. Therefore, we will implement our own triangulation algorithm and see how it compares to the GLU Tesselator.

Moving from 3D to 2D

To implement a triangulation algorithm that works for the faces of the grid cells is not a straightforward task. To make the triangulating simpler, algorithms are based on triangulation of polygons in 2D space [2]. Since the faces of the grid cells are 3D polygons, the first thing we have to do is to project the three-dimensional polygons into two dimensions. We will project the polygon into either the xy, xz, or yz plane. Care must be taken when projecting the polygon into of the mentioned planes. If for example one of the coordinates of the polygon's vertices is constant we cannot project the polygon into a plane using this coordinate, as that would reduce the polygon to a line. To avoid this problem, we project the polygon by computing its normal, and discard the coordinates corresponding to the coordinate with greatest magnitude in the normal. This will often, but not always, be equivalent to projecting the polygon into the plane where it has largest area [2]. The triangulation algorithm will need to create a separate data structure to work on, and the vertex data will therefore need to be copied. When copying the vertex data we modify it by computing the average position of the polygon and then subtracting this from all vertices. This is actually done before we compute the polygon's normal. The x and y coordinates of the grid will typically be very large, and we avoid some numerical instability issues by moving the polygon to the origin.

6.3.2 Ear clipping algorithm

When the polygon has been projected into two dimensions, we are ready to triangulate it. All polygons, convex or concave, can be triangulated by introducing diagonals [21]. A diagonal of a polygon P is defined as a line segment between two vertices a and b that does not intersect any of the polygon's boundary edges [21]. A naive algorithm based only on finding diagonals will be very inefficient and give a complexity of $O(n^4)$ [21]. This can be slightly improved to give an $O(n^3)$ algorithm. This is, however, still rather inefficient. Therefore we will use a triangulation algorithm called "Ear Clipping Algorithm" which has complexity $O(n^2)$. Three consecutive vertices of a polygon a, b, c are said to form an *ear* if ac is a diagonal. The algorithm starts by checking and marking the ear status of every vertex. A vertex v_i is an ear tip if the vertices v_{i-1} and v_{i+1} form a diagonal. Then the algorithm iterates over the polygon looking for vertices marked as ear tips. When an ear tip v_i is found, we know that v_{i-1}, v_i, v_{i+1} form a triangle that can be added to the list of triangles. The vertex v_i is then

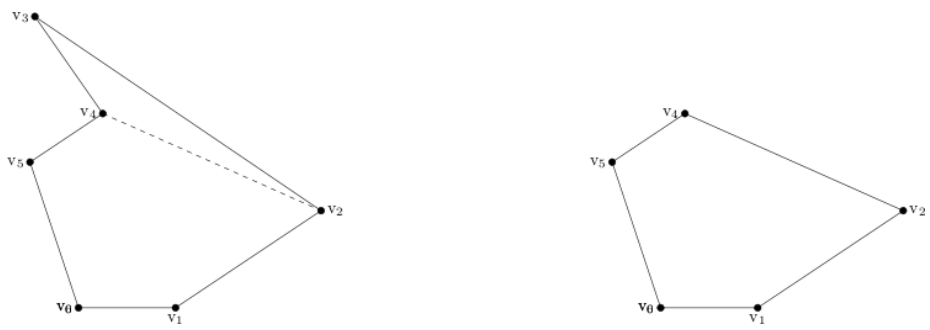


Figure 6.2: One step in the ear clipping algorithm. Vertex $\mathbf{v_3}$ can be removed since the line segment $\mathbf{v_2v_4}$ is a diagonal. After removing $\mathbf{v_3}$, the ear tip status of $\mathbf{v_2}$ and $\mathbf{v_4}$ is updated. Both will move from not being an ear tip to being an ear tip.

removed, and the ear tip status of v_{i-1} and v_{i+1} is updated. Eventually, all ear tips will be removed and the polygon is triangulated. To make the algorithm more efficient we introduce some cut-offs. If the polygon already is a triangle, there is no need to start the algorithm. If the polygon is a quadrilateral we can triangulate the polygon manually by finding a convex node. We can then decide how to triangulate the polygon by checking if one of the node's neighbors is concave or not. See Figure 6.2 and Algorithm 1.

Even though we transform the polygons to the origin before triangulating, we still encounter faces that are problematic to triangulate. This has to do with the geometry of the reservoir grid. A typical grid will be "long and flat", meaning that the difference between minimum and maximum z-coordinate will be much smaller than the difference in x- and y-direction. This will sometimes lead to faces that are so long and thin that the triangulation algorithm will interpret them as lines. To avoid an infinite loop, Algorithm 1 will jump out of the outer while-loop if it has inspected all the vertices without finding a vertex marked as an ear tip.

6.3.3 Rendering the boundary of the grid

Even though the `plot_grid` function will receive all the information describing the grid instead of just the boundary, more often than not, we will only be interested in rendering this boundary, as we did when rendering grids by using the alternative `patch` function. There are of course exceptions, for example when rendering the grid with a transparent boundary or zooming into the grid, but unless we are specifically interested in the interior of the grid, it would be a waste of resources to render the entire grid. The default behavior of the `plot_grid` function is therefore to only render the exterior boundary of the grid. We let OpenGL, or more precisely, the geometry shader, pick out the exterior boundary of the grid by discarding the interior of the grid. The way to decide if a face is

Algorithm 1 Ear clipping triangulation algorithm for triangulating simple polygons

```
vertices is a double linked list with nodes of the polygon
if vertices == 3 then
  Vertices already form a triangle, exit algorithm
end if
if vertices == 4 then
  Quadrilaterals can be triangulated more efficiently manually, exit algorithm
end if
for i = 0; i < vertices.size; i++ do
  if is_diagonal(vertices[i-1], vertices[i+1]) then
    mark vertices[i] as ear tip
  end if
end for

while vertices.size > 3 do
  for i = 0; i < vertices.size; i++ do
    if is_ear(vertices[i]) then
      add triangle to the list of triangles
      update ear tip status vertices[v-1]
      update ear tip status vertices[v+1]
      remove vertices[i]
      break
    end if
  end for
end while
```

a boundary face or not is to see how many cells the face is connected to. If the face is connected to two cells it is an inner face, whereas if it is only connected to one cell it is a boundary face. This neighboring information is found in the field `cells.faces` and needs to be made available for the geometry shader. However, since the geometry shader is executed once per primitive, which in our case will be a triangle, the data in `cells.faces` needs to be duplicated in the triangulation process, and then stored on the GPU. The indices of neighboring cells (one or two per triangle) are stored in a buffer and read through the texture unit on the GPU. This texture buffer is set up in such a way that the geometry shader, which receives one triangle, can look up in the texture to determine if the triangle belongs to one or two cells.

6.3.4 Rendering lines

When we have rendered the boundary of the grid by drawing the triangles, it is necessary to add lines around the faces of the cells to better see the geometry. Since the data now residing on the GPU is triangle based we cannot directly draw lines around all the triangles, as this would produce too many lines, see Figure 6.1. When we are triangulating the grid before sending the data to the GPU, we map information about cell neighbors from faces to triangles. At this point it is possible to mark the edges of a triangle that coincide with the edge of a face so that these can be drawn as lines.

We want to be able to plot as large grids as possible, and because memory on the GPU is a limited resource, we want to be as memory efficient as possible when storing this information. Therefore, we want to avoid using a separate buffer to hold the line information. The geometry shader has access to the texture buffer with the indices of neighboring cells. As discussed above, this information is divided up on a per triangle basis on the CPU and stored as unsigned integers before the information is moved to the GPU. The geometry shader has the possibility to change geometry. Therefore, if we know which edges of an incoming triangle should be drawn as lines, the geometry shader can transform the incoming triangle to one or more lines. A possibility for giving the necessary information to the geometry shader is to use the two leftmost bits of the cell indices belonging to each triangle to hold the information about which edges of that triangle should be drawn as lines. If two bits are used to hold edge information we are of course limiting the number of possible cell indices. A unsigned integer in the GLSL uses 32 bits. If two of these bits are occupied we are left with 30 bits, which means that we support grids with maximum $2^{30}-1$ $(1,073,741,823)$ cells. We will however run into other memory limitations long before this will become a problem. For a triangle in general there are six possibilities to which of the edges that coincides with a boundary of a face. However, due to how we triangulate the faces we can reduce the number of possibilities down to four, and therefore use only two bits by taking care of how we store the triangles. The bit pattern we will use is as following, where

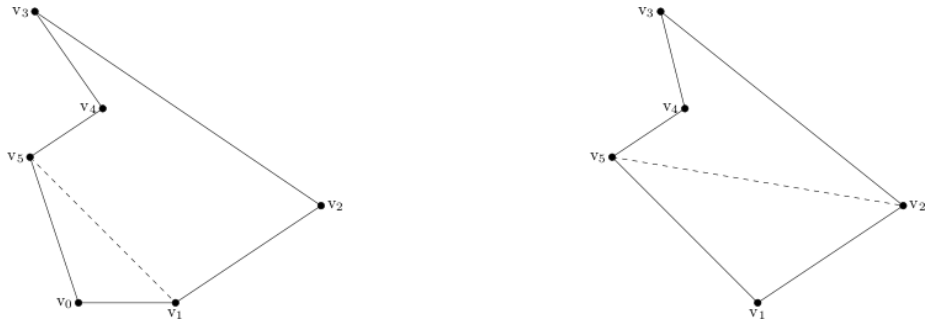


Figure 6.3: Illustration of how to decide which edges of a triangle should be draw as lines. When removing v_0, the line from v_5 to v_0 should be drawn, but when in the next iteration we remove v_1, the line segment v_5v_1 should not be drawn, since it does not coincide with the boundary of the face

the X's can be used to represent numbers between $2^{30} - 1$.

$$
\begin{aligned}
00XX\ldots XX &\rightarrow \text{None of the edges should be drawn} \\
01XX\ldots XX &\rightarrow \text{First edge should be drawn} \\
10XX\ldots XX &\rightarrow \text{First and second edge should be drawn} \\
11XX\ldots XX &\rightarrow \text{All edges should be drawn}
\end{aligned}
$$

How we decide which edges of a triangle to draw as lines is best illustrated by an example. Consider the polygon in Figure 6.3. Before we start to triangulate the face each node is marked with a boolean flag. If this flag is true it means that the line from this node to the next should be drawn as a line. When we start the algorithm we will start by removing v_0. The triangle $v_5v_0v_1$ should then be added to the triangle list. Since both v_5 and v_0's flags are true we want to draw the two line segments v_5v_0, v_0v_1 as lines. This means that we set the two leftmost bits in the array corresponding to this triangle's neighbor information to 10, which means that the first and second edge of this triangle should be drawn as lines. Now that we have stored a triangle where the edge from v_5v_0 is drawn, we must remove v_5's boolean flag. In the next iteration of the algorithm we will remove v_1. However, since v_5's flag is false we only want to draw the second edge of the triangle $v_5v_1v_2$. When only using two bits we do not have any pattern for drawing only the second edge. The solution is to change how the triangle is added. Instead of adding the triangle $v_5v_1v_2$ to the list of triangles we add the triangle $v_1v_2v_5$ and use the bit pattern for drawing the first edge of a triangle (01). By changing the order of how the indices are added, the four permutations of the two bits allow us to describe all possibilities.

6.3.5 Using transform feedback

When rendering the grid we have two OpenGL draw calls: One call to draw the triangles of the grid, and one call to draw the lines around the edges of the

cell's faces. Two different geometry shaders are used for these draw calls. The geometry shader associated with the first draw call makes sure only the relevant triangles are drawn. The second geometry shader only passes on lines that coincide with the edges of the faces. We end up doing quite a bit of work in the two geometry shaders, which naturally will effect the performance of the grid rendering. After the grid has been rendered the first time, and the geometry shaders have picked out the triangles and lines that we want to render, we do not really need the computation to be performed again if we only want to do operations that do not deal with the interior of the grid. Therefore we make use of an OpenGL feature called *transform feedback* that allows us to save the results from the vertex, or geometry shader, into a buffer object [13] and then render from this buffer. This is all done on the GPU without transferring any data to the CPU. By using transform feedback to capture the results from our two geometry shaders into two separate buffers, we will not need to run the geometry shaders in subsequent draw calls, as we then have the triangles and lines that form the boundary of the grid stored in buffers. The contents of these buffers can be drawn at a much higher frame rate since we omit the geometry shaders.

6.3.6 Cut planes

Often when inspecting a reservoir model visually we would like to take a closer look at the inner structure of the reservoir. This could be, for example, to look at what happens to the model along a fault. By writing Matlab/Octave scripts, this is possible to do in MRST. However, then the computations will be performed in MATLAB or Octave, and will be fairly slow for large grids. In addition all the geometry will be have to be sent to the GPU for each cut plane. It would therefore be beneficial to keep the geometry on the GPU and compute which parts of the grid to visualize based on where the user wants to draw a cut plane. The performance differences between doing the computations in Matlab and Octave, and on the GPU using OpenGL, will be discussed in the next part of the thesis. This section will deal with the implementation details for supporting cut planes in the `plot_grid` function.

Representing a cut plane

We need some way to represent the cut plane and choose which parts of the grids that are relevant based on this representation. A simple way to do this is to use a plane equation. A plane can be defined by a point on the plane $\mathbf{x}_0 = (x_0, y_0, z_0)$ and the plane's normal vector $\mathbf{n} = (a, b, c)$. A point $\mathbf{x} = (x, y, z)$ is on the plane if

$$\mathbf{n} \cdot (\mathbf{x} - \mathbf{x}_0) = 0$$

This can be written on component form to yield the general plane equation

$$ax + by + cz + d = 0, \qquad d = -ax_0 - by_0 + cz_0$$

Using this equation we can easily determine if a point is on the plane, or on either of its sides. If a point inserted into this equation results in a number greater than zero it is on the positive side of the plane, which is defined as the side the normal vector points to. Similarly, a point is on the negative side of the plane if the result of inserting the point into the plane equation is negative.

Selecting cells based on a plane equation

As soon as we have the plane equation, we are ready to pick out the relevant part of the grid and visualize it. The relevant part of the grid will be the subset of the cells that are on the positive side of the plane provided. We define a cell to be part of the subset if one or more of the corners of the cell are on the positive side of the plane. To pick out the correct subset we need to inspect all the cells, and in some way mark the relevant cells. The inspection of a cell is independent of the inspection of all other cells, hence this task is well suited for being performed in parallel. Modern GPUs have hundreds of cores and are specifically designed for running a very large number of tasks in parallel. It seems quite evident that we should let the GPU select each of the cells, especially since a crucial part of the cell description already is located in a buffer on the GPU, namely the coordinates of the nodes. There are several ways to instruct the GPU to perform the inspection of all the cells. Both CUDA and OpenCL support interoperability with OpenGL, and are possible candidates to solve the task at hand. However, since we already are in an OpenGL context, the most natural is to do the computations using OpenGL.

The idea then is to execute a vertex shader that will run the same number of times as there are cells in the grid. The vertex shader compares the coordinates that make up the cell, to the plane equation, and determines if any of the nodes are on the positive side of the plane. It outputs the integer 1 if the cell is part of the subset and 0 otherwise. This output is recorded using transform feedback, and after all the cells have been processed we are left with a buffer on the GPU containing the same number of elements as there are cells in the grid. However, before the vertex shader can be executed it needs to be able to access all the information necessary for each grid cell. As exemplified in Section 3.3, we need access to the fields `facePos`, `faces`, `nodePos`, `nodes` and `coordinates` in the MRST grid representation to find the coordinates of the nodes a cell is made up from. As earlier mentioned, the `coordinates` field is already on a buffer on the GPU, since the coordinates are used when first rendering the grid, but the other fields must be transferred to the GPU and stored in buffers. Then we attach them to texture objects to be able to access the fields in the vertex shader using `texelFetch`. Even though the vertex shader is started by performing a draw call that is guaranteed to make the shader execute once for each cell, we have no wish of rendering any of the results from the computation. OpenGL has the possibility to turn off the rasterization process, meaning that the vertex shader will run, but the OpenGL pipeline will be chopped off after it has been completed [13].

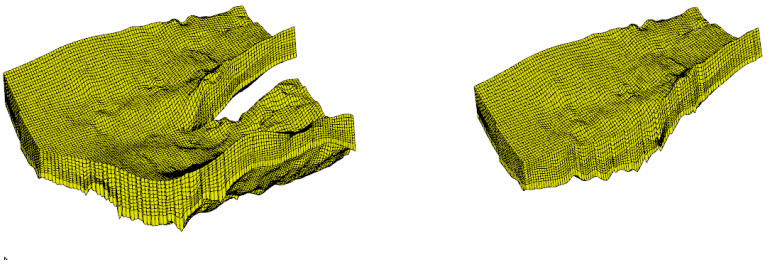


Figure 6.4: Illustration of the Johansen formation, complete model and with a cut plane

Rendering a subset of the grid

When all the cells have been inspected we are left with a buffer on the GPU where the element at position i denotes if global cell number i is part of the subset to be rendered or not. Rendering the correct part of the grid is now a matter of restarting the regular render loop with a slight modification. The geometry shader, which is responsible for extracting the boundary of the grid, needs to know which cells that should be rendered, and must therefore be able to access the buffer produced by inspecting the cells. Unless we have a transparent boundary, or the inner geometry of the grid otherwise is of interest, we still do not need to use resources to plot anything but the boundary of the grid. The geometry shader's criteria for choosing to draw a triangle or not will now have to change. A triangle should now be drawn if one neighbor is in the subset and the other is not, or if the triangle is on the boundary and belongs to a cell that is part of the subset.

6.3.7 Implementing the `plot_grid` function in Octave

When implementing the `plot_grid` method in Octave we take the same approach as with the `patch2` method from Section 6.2.2. We encapsulate all the OpenGL functionality and geometric processing in a class, and keep a STL map in the `opengl_renderer` class with integer values as keys and pointers to the grid class as values. We also introduce a new method in the `opengl_renderer` class

C++ code

```
void
opengl_renderer::draw_unstructured_grid (const
    grid::properties &props);
```

The argument to this function will be given its unique integer identification from the Octave interpreter.

With the `plot_grid` method we are faced with a new challenge: How to let the user interactively choose a cut plane? For the new functionality this

method offers to be useful in practice, we must make it easy for a user to change the cut plane. We can of course have the user send a cut plane from the Octave interpreter, but that would not be real interaction. A better solution would be that the user chooses a plane, and can move this plane with either mouse or keyboard. To enable this feature in Octave, we would need to send information from the FLTK part of the program when a graphics object of type grid is rendered down to the `draw_unstructured_grid` function, located in the `opengl_renderer`. Octave is not designed for this kind of information flow, and although it is possible to alter, it would involve changing large parts of the existing code base. We would like to limit ourselves to extending the existing code, as this makes it much easier to integrate the methods in future versions of Octave. This is an important argument, since Octave is a project under continuous development. Thus, since communication with a user is difficult to achieve, we also implement the `plot_grid` method in a standalone application that uses *freeglut* [3] as window toolkit. Octave will communicate with the standalone application through an oct-interface. This usage of oct-files is not standard, and is used only as an illustration of what we can achieve when freeing ourselves from the Octave framework, since a full implementation of a separate window toolkit is outside the scope of the thesis. Another reason for implementing the `plot_grid` in a standalone application is that, as we will see in Chapter 7, it will give us a performance benefit. In the same chapter we will present functionality of the standalone `plot_grid` method.

Part III

Results and Discussion

Chapter 7

Results

It is time to test the various methods we have developed. This will be done in two parts: First we will test functionality, and then performance. In both parts of the tests we will compare against Octave's standard implementation and Matlab. In the first part of the testing we will focus on comparing the new implementation of `patch` against Matlab and Octave. The `plot_grid`, having a usage a bit on the side of the other methods, will be tested separately. All methods will be tested equally during the performance part of the test.

7.1 Functionality

To test the functionality we have carried out certain parts of some of the tutorials from the MRST project's homepage [28] which illustrates what the methods accomplish, and what they do not accomplish.

We will start out by visualizing the result of a simple pressure solver on an idealized, cubic grid from the *Gravity Column* tutorial. The following code has been run for both implementations of `patch` in Octave and in Matlab.

```
% Set up the model with properties and boundary conditions
gravity reset on

G           = cartGrid([2, 2, 30], [1, 1, 30]);
G           = computeGeometry(G);
rock.perm   = repmat(100*milli*darcy(), [G.cells.num, 1]);
fluid       = initSingleFluid('mu' ,    1*centi*poise, ...
                              'rho', 1000*kilogram/meter^3);
bc   = pside([], G, 'TOP', 100.*barsa());

% Assemble and solve the linear system
T    = computeTrans(G, rock);
```

```
sol = incompTPFA(initResSol(G, 0.0), G, T, fluid, 'bc', bc);

% Plot the face pressures
newplot;
plotFaces(G, 1:G.faces.num, convertTo(sol.facePressure, ...
            barsa()));
set(gca, 'ZDir', 'reverse'), title('Pressure [bar]')
view(3), colorbar
```

The only plotting command used is `plotFaces`, where the pressure is given as the color argument. From Figure 7.1, we see that there is no real differences in the way the grid is visualized.

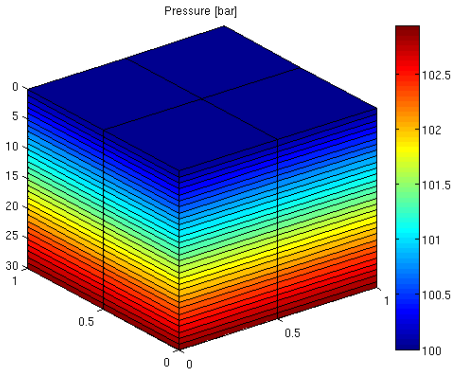
We then move on to visualizing a grid model of a part of the Johansen formation. The following code has been run for both implementations of `patch` in Octave and in Matlab

```
sector        = 'NPD5';
grdecl        = readGRDECL([sector, '.grdecl']);
actnum        = grdecl.ACTNUM;
grdecl.ACTNUM = ones(prod(grdecl.cartDims),1);
G             = processGRDECL(grdecl, 'checkgrid', false);

% Plot the grid, mark the faults with red color
clf
plotGrid(G,'FaceColor','none','EdgeAlpha',0.1);
plotFaces(G,find(G.faces.tag>0),'FaceColor','r');

% Mark the active part of the grid in blue
plotGrid(G,find(actnum(G.cells.indexMap)), 'FaceColor', 'b',...
          'FaceAlpha', 0.4, 'EdgeAlpha', 0.1);
view(20,75);
```

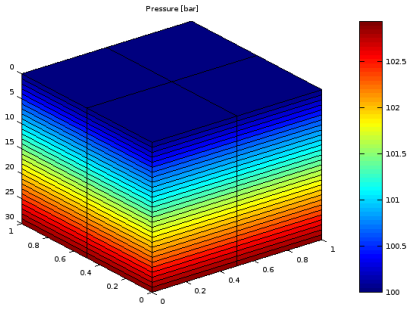
The results are seen in Figure 7.2. Octave's original `patch` function does not at all produce the desired result. The problem lies in the use of *EdgeAlpha* in the two `plotGrid` commands, which is not accepted by Octave's `patch` function, the result is that nothing is produced when the *EdgeAlpha* option is used, so we only get a plot of the faults. The new implementation of the `patch` function, however, produces a result almost identical to the Matlab plot.

We continue with the Johansen formation, but now we plot a height map of the active cells.

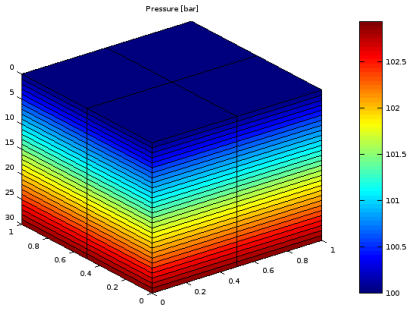
```
sector        = 'NPD5';
grdecl        = readGRDECL([sector, '.grdecl']);
G = processGRDECL(grdecl); clear grdecl;
G = computeGeometry(G);

% Plotting a height map of the field using the
```



(a) Matlab

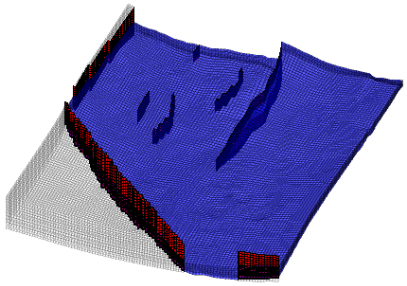


(b) Octave (`patch`)



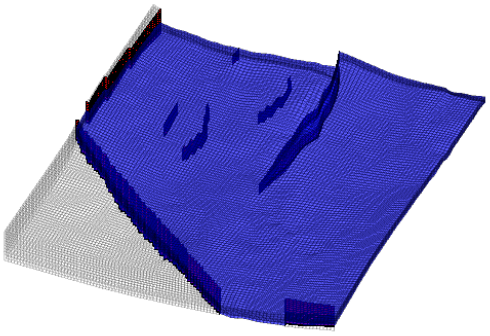
(c) Octave (`patch2`)

Figure 7.1: Visualization face pressures.



(a) Matlab

(b) Standard Octave rendering (`patch`)



(c) Improved Octave rendering (`patch2`)

Figure 7.2: Visualization of the Johansen formation with faults.

```
%z-component of the centroids of the cells
clf,
plotCellData(G,G.cells.centroids(:,3),'EdgeColor','k',...
             'EdgeAlpha',0.1);
colorbar, view(3), axis tight off, view(-20,40), zoom(1.2)
```

The plots with the three different functions are presented in Figure 7.3. In this case, the plot from Octave's `patch` function is usable, though not completely correct, compared to the Matlab plot. Once again, the problem is the use of *EdgeAlpha*, which results in the edge lines not begin drawn. The plot produced by the new implementation of `patch` is very similar to Matlab's plot. Both Octave plots slightly differ from the Matlab plot in shape. This is because the `zoom` function is not yet implemented in Octave.

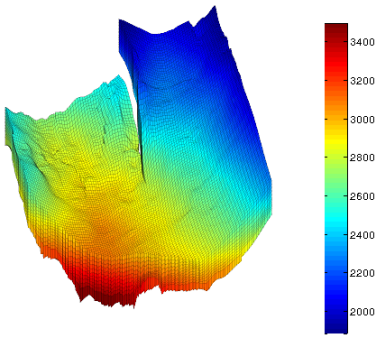
7.1.1 Functionality of the `plot_grid` function

The functionality of the `plot_grid` function differs from the other functions, and we therefore test it separately. It is not meant as a replacement for the other functions tested above, but as an example of more advanced visualization functionality. As we saw in in Chapter 6, it is difficult to implement different user interactions than what Octave's toolkit already supports, so the `plot_grid` function has been implemented in a standalone application which Octave communicates with through the use of an oct-file.

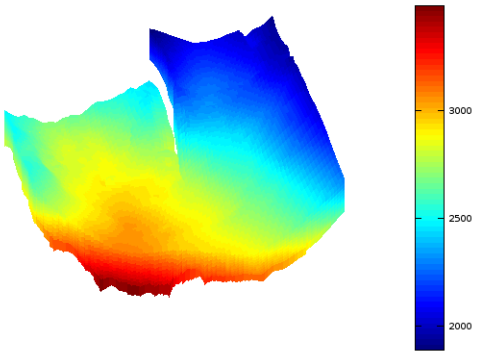
The standalone implementation of the `plot_grid` function allows the user to choose one of the coordinate planes (the xy-, yz-, or xz-plane) and control its position interactively with the mouse cursor. This can be useful when visualizing a grid model with a certain property, to get a better impression of how this property changes inside the grid.

As an example we have plotted the porosity data of each cell in the Johansen formation:

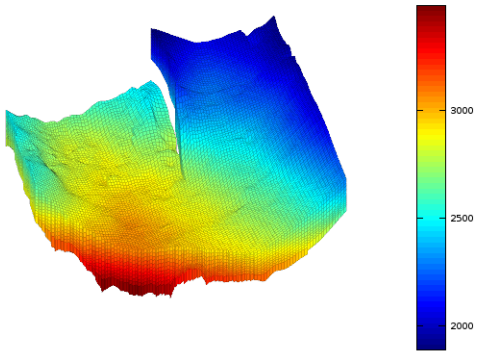
```
% Read the grid model
sector        = 'NPD5';
grdecl        = readGRDECL([sector, '.grdecl']);
G = processGRDECL(grdecl);
G = computeGeometry(G);

% Read porosity data
p = reshape(load([sector, '_Porosity.txt'])',...
            prod(G.cartDims), []);
rock.poro = p(G.cells.indexMap);

% Call the Octave function standalone_plot_grid
% with the porosity data as color values.
% This function uses an oct-interface to
% send data to and launch the C++ application
standalone_plot_grid(G, 'CellColor', rock.poro)
```



(a) Matlab



(b) Octave (`patch`)



(c) Octave (`patch2`)

Figure 7.3: Visualization of a height map of the Johansen formation.

The results can be seen in Figure 7.4. By interactively controlling a cut plane it is easy to see porosity values of the interior of the grid.

We can achieve something similar when using one of the MRST plotting functions, but then only through written instructions, which makes it more difficult to use.

7.1.2 Summary of functionality tests

The three examples have been chosen to illustrate the trend of the methods. MRST often makes use of the *EdgeAlpha* option to make the edge lines less prominent. This is most easily done by reducing the edge alpha, since there is no easy way in Matlab to reduce line width. Octave does not provide any way to change line width, and in addition it is not possible to control the visibility of the lines with *EdgeAlpha* when using the original `patch` function. This feature is not crucial, as it is possible to render lines with the original `patch` function if we avoid the use of *EdgeAlpha*. Nevertheless, it is a lacking feature in the original `patch` function. The problem of not being able to control *FaceAlpha*, however, is much more severe. This makes Octave's `patch` function useless for plotting both the outline and inner parts of grids, something that is not uncommon in MRST. Octave's `patch` function is therefore only usable in the simplest cases. The new implementation of the `patch` function offers functionality almost equal to that of Matlab. There are some slight differences, but these will often not be crucial for the usage. The `plot_grid` function implemented in a standalone application introduces new functionality that cannot be matched directly by Matlab.

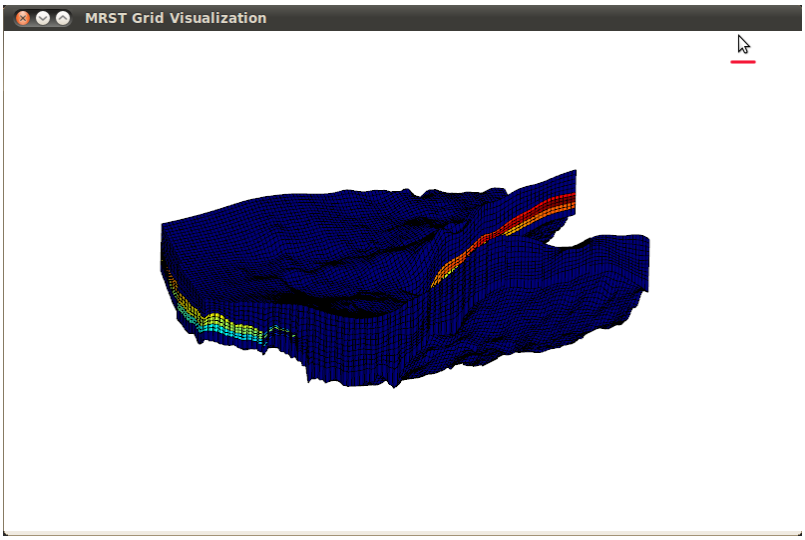
7.2 Performance

The functionality part of the testing is only part of the story. All the grids are three-dimensional, and to inspect the entire grid with its properties, we must be able to inspect it from all sides. This is best done by interactively rotating the grid. We have earlier mentioned that Octave quickly becomes almost useless as the grid size increases, because of its low frame rate. We experience the same behavior in Matlab, although it works better than Octave. If the newly developed methods are able to improve this behaviour it would be an argument for using Octave instead of Matlab. We will now test the performance of the methods developed in Chapter 6 and compare them against each other, standard Octave, and Matlab. To do this, we will visualize grids of different size and apply rotations and cut planes.

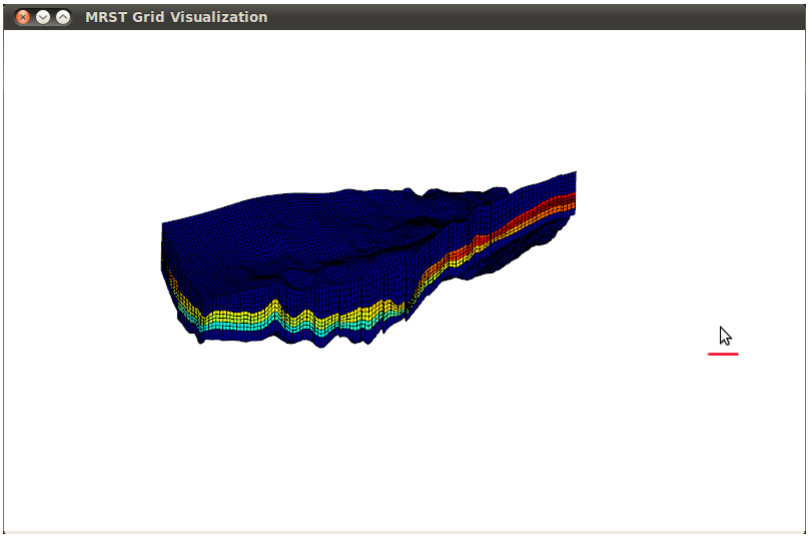
7.2.1 Test hardware

All the methods have been tested on a system running Ubuntu 10.04 with the following specifications:

- NVIDIA GeForce GTX 480 (1536MB)



(a) Full grid



(b) Grid with cut plane

Figure 7.4: Porosity plot of the Johansen formation which illustrates interactive use of cut planes. The vertical position of the mouse pointer (underlined in red) controls the cut plane.

- Intel Core i7 2.67 GHz
- 6 GB Memory

The system uses driver version 295.33 for the graphics card.

7.2.2 Benchmarking

To compare the different methods against each other, we have performed a series of tests on various grids. As the main criteria of comparison we have chosen to look at how many frames per second (fps) the different methods produce on grids of various sizes. In addition to enabling us to compare the methods, this will also give us a good idea of how well it is possible interact with the grids for a user. This means that our measurements will include some of the overhead produced by Octave since the tests have been performed from Octave.

We will get an idea of how much overhead Octave imposes by looking at the results from the standalone implementation of the `plot_grid` function. This function also has been implemented in Octave and by comparing the results from the standalone application with the Octave implementation we will see how Octave affects the frame rate.

The tessellation of the grid is a central part of the processing that needs to be done before the grid can be rendered. Since the functions use different tessellation methods we have also measured the time the `GLU Tesselator` and the ear clipping algorithm use to tessellate the different grids.

7.2.3 A comment on frame rates

Since we will be using frame rate as a measure on how well it is possible to interact with the grid it is necessary to define what we mean by an acceptable interaction frame rate. If the frame rate is too low, it will be difficult to control the grid when rotating it. The human eye and its data reception and transmission system can analyze 10-12 images per second [23], so a frame rate higher than that will result in a sensation of visual continuity. The standard frame rates in video and TV making varies from 24-30 frames per second. We do, however, not need the same visual quality as TV or video for the grid to be possible to work with. We suggest that a frame rate of 15-20 fps is a minimum for interaction with a grid model.

7.2.4 Methods tested

The methods we are going compare against each other are: Octave's original `patch` function, the alternative `patch` function, the `plot_grid` function, and Matlab's `patch` function. The `plot_grid` method will be tested both in a standalone application and integrated into Octave. This will help shed light on the overhead we get by Octave's framework.

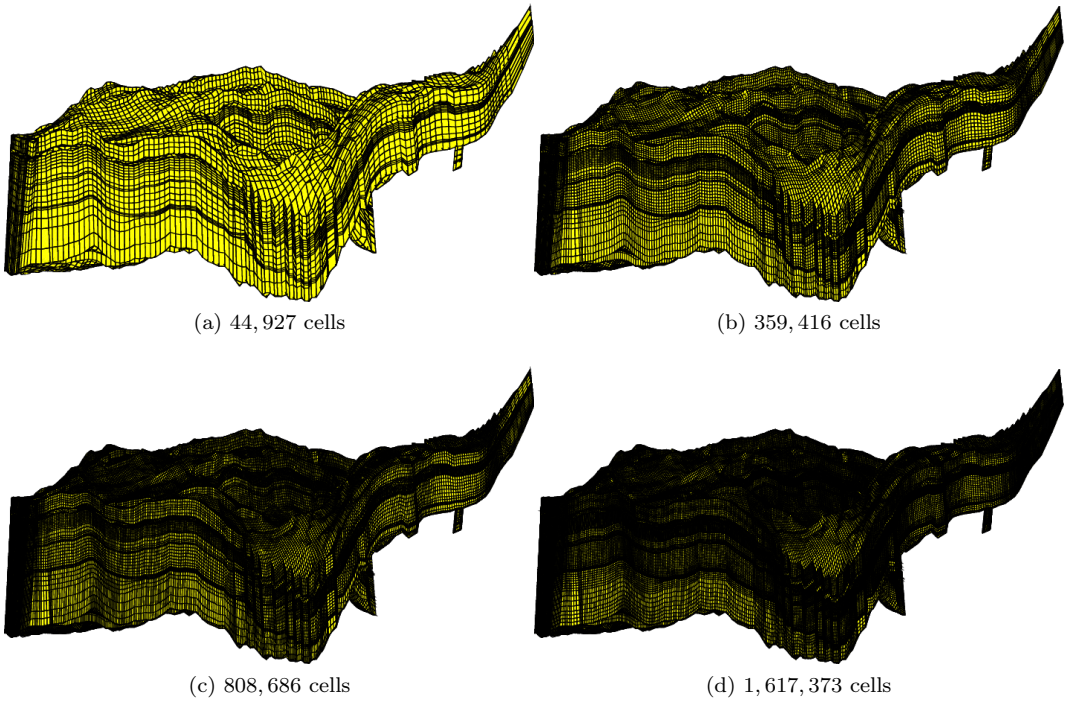
To test the methods, we will test how much time it takes for each of them to visualize a grid. We will also see how much time it takes to extract, and plot, a subset of the grid. To use the `patch` functions to visualize a grid, we need to use one of MRST's plotting functions, see Figure 3.9. All of these methods eventually use `patch` to render the grid, but they differ in what they accept as arguments. We have decided to use two different MRST functions: `plotFaces` and `plotGrid`. The reason we have chosen two MRST methods is that `plot_grid` works a bit differently than the MRST functions, and it is difficult to find a single function that will give a fair comparison. A good reason for using the MRST function `plotFaces` is that then the `patch` functions and `plot_grid` would receive the same amount of data from Octave, so that the same amount of data would be sent through the graphics pipeline. However, as described in Chapter 6, the `plot_grid` function discards inner geometry, so that only the exterior boundary of the grid is rendered. The first time the grid is rendered with the `plot_grid` function the entire grid will pass through the graphics pipeline, but subsequent times, a reduced amount of data will pass through the pipeline. Therefore, we will also use the MRST function `plotGrid` for comparison. This function takes the entire grid as argument and then extracts the boundary faces before sending any data to the C++ implementation. The boundary faces, represented as vertices and an index list, are then sent to the `plotFaces` function which calls `patch`. The amount of data processed by the underlying C++ implementations will therefore not be the same for the `patch` functions and `plot_grid`, but it will be a more realistic comparison in terms of how a user would utilize the functions.

7.2.5 Test setup

To test how the functions perform, they have been used to render a grid representing a real model that we have sub-sampled three times to get sufficiently large data sets. The four grids have approximately $50,000$, $350,000$, $800,000$, and $1,600,000$ cells, see Figure 7.5.

For each grid we have used the different functions to render them a large number of times, and then computed the average frame rate. We have run each test two consecutive times, where the first run has been a dry run to prepare the graphics driver for the type of data it will receive. To ensure that the entire grid is redrawn for each frame, we have applied a random rotation matrix for every iteration.

The tests are designed so that we get an impression of how well the different methods would perform in a normal work situation, where the methods will be exposed to different workloads. We have therefore chosen a set of modification ratios for how often a new half plane is selected, and a subset of the grid is computed and rendered. The modification ratios are 1:1000, 1:100, 1:10, 1:2, and 1:1. The lowest ratio is a measurement of how well the methods perform when the grid size is nearly static, and tests the pure rendering speed of the method. This could have been measured by not selecting any half planes at all, but then the amount of data to be rendered would be much larger than



(a) 44,927 cells (b) 359,416 cells

(c) 808,686 cells (d) 1,617,373 cells

Figure 7.5: A model from the Norwegian sea, same grid with different resolutions.

for the other modification ratios, since they will render a subset of the grids. When we only choose a new subset of the grid every thousand iteration the time taken to select a subset will be negligible compared to the time needed to render. The two ratios 1:100 and 1:10 simulate what would probably be the most common work flow, where a user would inspect the grid while trying to find the correct subset to render. The two highest ratios measure how well the methods performs when constantly selecting new half planes.

The testing and timing of all the functions have been performed by Octave, except the testing of the standalone application. The command `view` has been used for every iteration to change the viewpoint of the axis. A random half plane has been computed to pick out a subset of the grid. The half plane goes through the grid's midpoint, and the plane's normal vector varies in all directions. The `plot_grid` function accepts a half plane equation as argument. The MRST functions `plotFaces` and `plotGrid` accept no such argument. What they do accept, however, is a list of the global face numbers or global cell numbers to plot. Before calling the functions from Octave it is therefore necessary to find the correct face or cell list to send to the methods. To find these cells and faces

Table 7.1: Triangulation results measured in seconds.

Number of polygons	155,329	1,158,340	2,553,187	5,089,021
`GLUtesselator`	0.5	4.1	10.2	18.8
Ear clipping	0.1	0.9	1.8	4.0

we use the Octave functions shown in Appendix B.

7.3 Test Results

We will now present the results from each of the methods we have tested. We will start with the tessellation results, as they will be important to understand the other results.

7.3.1 Results for the tessellation algorithms

We make use of two different tessellation methods: `GLUtesselator` and the implementation of the ear clipping algorithm described in Chapter 6. Both the `patch` functions use the `GLUtesselator`, while the `plot_grid` function uses the ear clipping algorithm. To test the performance of the two tessellators we have tessellated the grids in Figure 7.5 with both methods a large number of times and averaged the results.

The triangulation results are presented in Table 7.1 and in Figure 7.6. The different number of polygons correspond to the different grids in Figure 7.5. The ear clipping algorithm is about five times faster than the GLU Tesselator. Figure 7.6 shows that the Ear Clipping algorithm yields linear complexity even though it has worst case complexity $O(n^2)$. From this we can conclude that the grids consist of a large percentage of triangles and quadrilaterals, and that the cut-offs we introduced in Section 6.3.2 are efficient. The algorithm used by the `GLUtesselator` is a custom-made algorithm, developed specifically to be as robust as possible [29]. It also offers linear complexity, but with a larger constant.

It is worth noticing that the `GLUtesselator` uses 0.5 seconds to tessellate even the smallest number of polygons measured, which correspond to the grid with approximately $50,000$ cells. This will severely impact the performance of the methods that use `GLUtesselator` every time the grid is rendered.

7.3.2 Results for `plotFaces`

The results for the tests performed by the MRST function `plotFaces` can be seen in Table 7.2. The `plotFaces` function takes all the faces of the grid as argument and passes the vertices and indices to one of the `patch` functions. Because of the very low frame rate we got when testing Octave's `patch` function

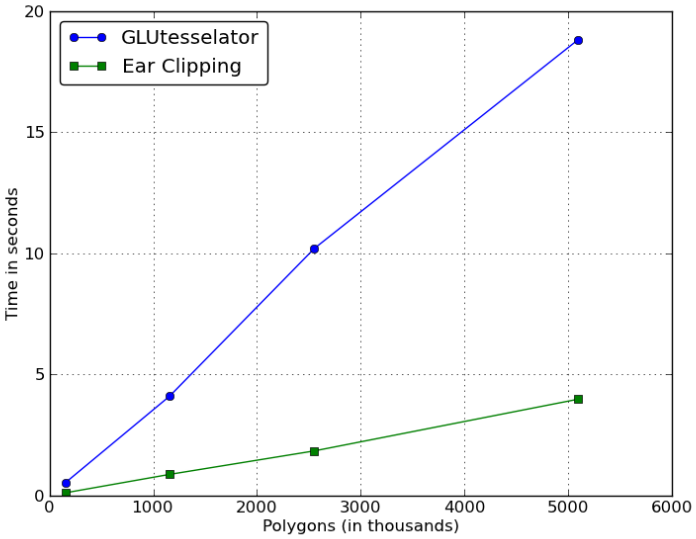


Figure 7.6: Performance results triangulation.

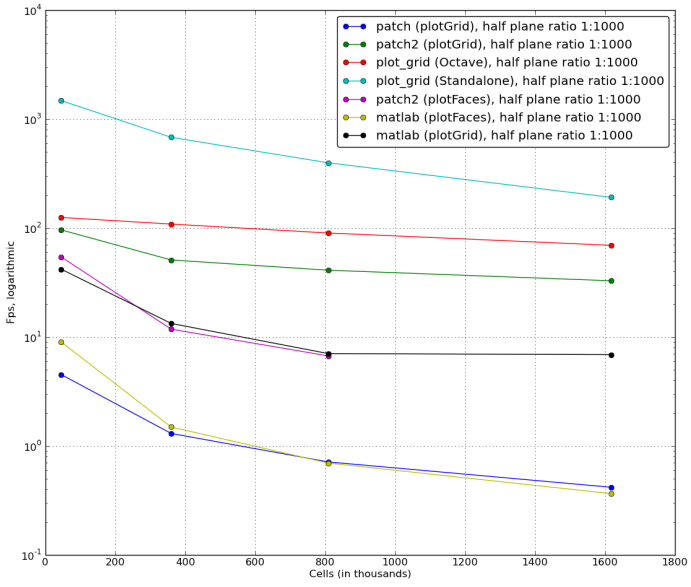


Figure 7.7: Performance results, 1:1000 mod. ratio.

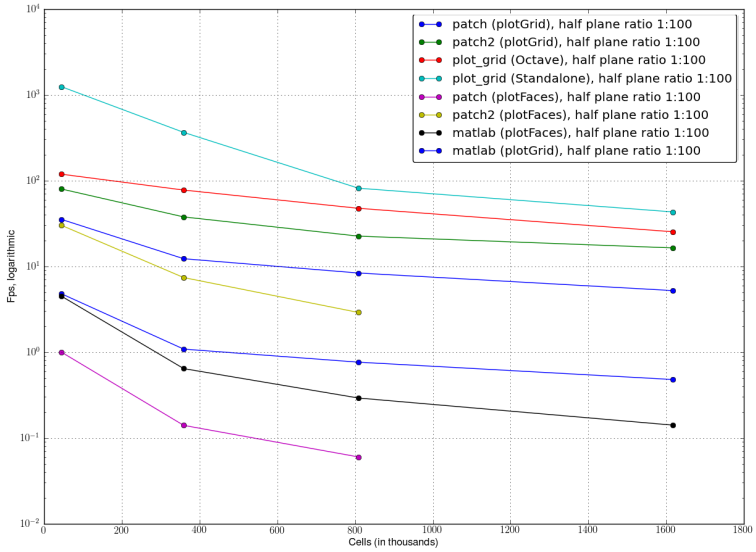


Figure 7.8: Performance results, 1:100 mod. ratio.

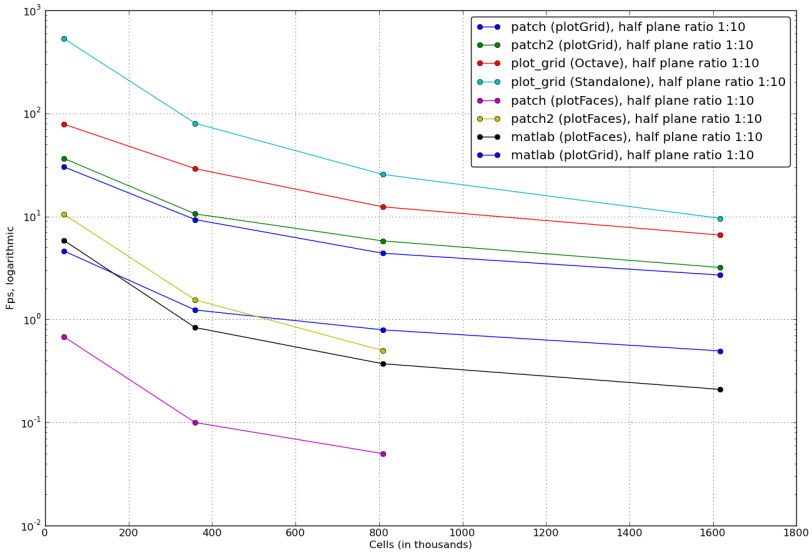


Figure 7.9: Performance results, 1:10 mod. ratio.

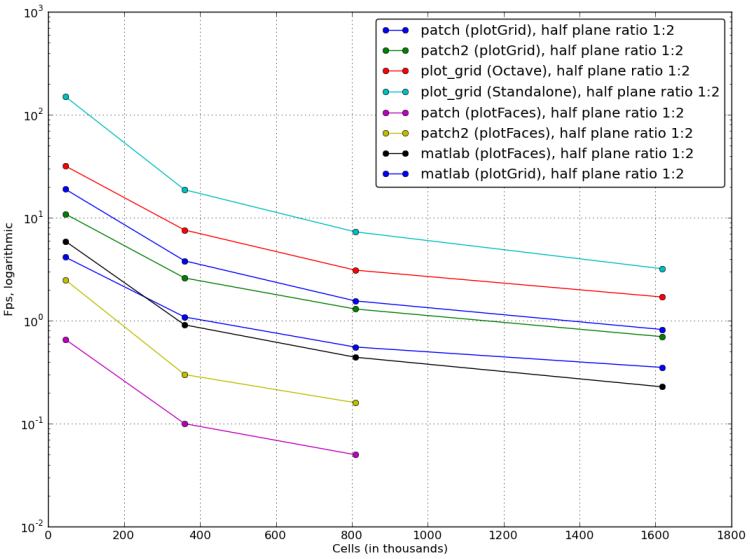


Figure 7.10: Performance results, 1:2 mod. ratio.

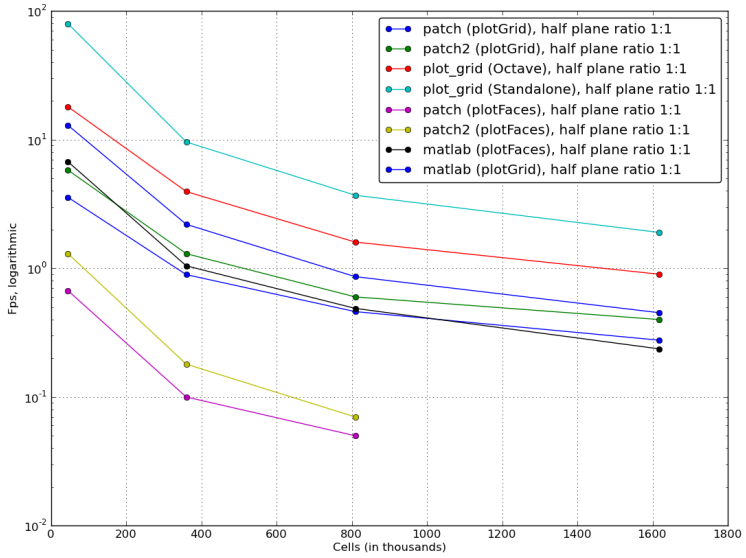


Figure 7.11: Performance results, 1:1 mod. ratio.

we skipped the 1:1000 modification ratio. It is also worth noticing that both Octave methods ran out of memory when trying to render the largest grid.

Octave's `patch` function is by far the slowest method. It is limited in frame rate by the `GLUtesselator`, which is used every time a frame is rendered. From Table 7.1, we see that when rendering the smallest grid when a half plane is applied, the frame rate when using Octave's `patch` function can be no more than four frames per second. This itself is not an impressing number, and makes interaction with the grid cumbersome. In addition the rendering and overhead imposed by Octave reduces the frame rate even more. It is worth noticing that when using Octave's `patch` function, the modification ratio does not affect the frame rate much. This means that the time used to actually find a subset of cells based on a plane equation is very small compared to the total rendering time.

Octave's `patch2` function performs much better than the original one, and also outperforms Matlab on the three lowest modification ratios. As the modification ratio increases there is a drop in the frame rate, and when we reach the highest modification ratio there is not much difference between `patch` and `patch2`. Both these methods then uses the `GLUtesselator` for every frame.

Matlab is less affected by modification ratio than the `patch2` function. From the results, it seems like Matlab has less efficient rendering than `patch2`, but uses a faster triangulation algorithm.

Figures 7.7 to 7.11 show the `plotFaces` results compared with the other methods. Octave's `patch` gives the lowest frame rate of all the methods tested. The `patch2` function has a frame rate similar to `plotGrid` in Matlab for the lowest modification ratios, but becomes the second slowest method when the modification ratio increases.

Generally, we see that there is a considerable speedup to be gained by tessellating the data only when necessary, and using more efficient rendering methods than immediate mode rendering. A typical work flow when using these methods will be to rotate the model without extracting subsets very often, since this is only supported by scripts. We see that the only method that allows easy interaction with the grid is the `patch2` method, and then only for the smallest grid.

7.3.3 Results for `plotGrid`

Table 7.3 shows the results of the tests where the MRST function `plotGrid` has been used. This function extracts the exterior boundary before sending data to the `plotFaces` function.

The trend in the results are very similar to results in the previous section. Octave's `patch` function has the lowest frame rate. Octave's `patch2` function has the highest frame rate for the three lowest modification rations, but becomes slower than Matlab for the two highest ratios.

The frame rates are, however, generally much higher when using `plotGrid`, as should be no surprise. The amount of data now processed and rendered is much smaller. Nevertheless the visual result is quite similar, as long as we

Table 7.2: Results for the `plotFaces` method measured in frames per second.

Method	Ratio	50k cells	350k cells	800k cells	1,600k cells
`Matlab`	1:1000	9.0	1.5	0.7	0.4
	1:100	4.5	0.6	0.3	0.1
	1:10	5.9	0.8	0.4	0.2
	1:2	5.9	0.9	0.4	0.2
	1:1	6.7	1.0	0.5	0.2
`patch`	1:100	1.0	0.1	0.06	-
	1:10	0.7	0.1	0.05	-
	1:2	0.7	0.1	0.05	-
	1:1	0.7	0.1	0.05	-
`patch2`	1:1000	54.2	11.8	6.7	-
	1:100	30.1	7.4	2.9	-
	1:10	10.5	1.6	0.5	-
	1:2	2.5	0.3	0.2	-
	1:1	1.3	0.2	0.07	-

do not zoom into the grid. Matlab now gives a frame rate that is acceptable for interacting with the grid when rendering the smallest grid with the lowest modification ratio. When the grid size increases, however, rotation of the grid will be much less smooth. When the modification ratio is low, Octave's `patch2` function has frame rates that allows for smooth interaction with all the grids we have tested.

In Figure 7.7, 7.8, and 7.9, we see that Octave's `patch2` function, when used by `plotGrid`, is only outperformed by the two implementations of `plot_grid`. In Figure 7.10 and Figure 7.11, Matlab and `patch2` swap places, making Matlab the method with third highest frame rate.

There is a considerable performance benefit when using Octave's `patch2` function instead of Octave's original `patch` function. Instead of getting a frame rate that makes interaction with even the smallest grids cumbersome, we get a frame rate that support easy interaction with even large grids, and that is notably higher than Matlab's plotting performance.

7.3.4 Results for `plot_grid`

The results from the testing of the `plot_grid` method are summarized in Table 7.4. This method takes the whole grid structure as argument, triangulates the data once using the Ear Clipping Algorithm, and does the rest of the work on the GPU. The method has been integrated in Octave, and implemented as a standalone application.

The `plot_grid` method benefits from faster tessellation than the other methods, as we see from Table 7.1. In addition, the tessellation is only performed once, as the method does not need to re-triangulate the data when choosing a

Table 7.3: Results for the `plotGrid` method measured in frames per second.

Method	Ratio	50k cells	350k cells	800k cells	1,600k cells
`Matlab`	1:1000	41.9	13.3	7.0	6.9
	1:100	35.3	12.3	8.4	5.2
	1:10	30.3	9.3	4.4	2.7
	1:2	19.0	3.8	1.6	0.8
	1:1	12.9	2.2	0.9	0.5
`patch`	1:1000	4.5	1.3	0.7	0.4
	1:100	4.8	1.1	0.8	0.5
	1:10	4.6	1.2	0.8	0.5
	1:2	4.1	1.1	0.6	0.4
	1:1	3.6	0.9	0.5	0.3
`patch2`	1:1000	96.4	50.9	41.0	32.8
	1:100	79.8	37.7	22.5	16.4
	1:10	36.6	10.6	5.8	3.2
	1:2	10.9	2.6	1.3	0.7
	1:1	5.8	1.3	0.6	0.4

subset of the grid to render.

The results show us that Octave imposes a rather big performance penalty compared to the standalone application. The frame rate in the standalone application is from 2.1 to 11.8 times higher than the frame rate in the Octave version. The reason for this difference is that the standalone application is much more light-weight than Octave. In Section 4.3.1, we described how Octave's plotting system is based on graphics objects. Each time a new frame is rendered the FLTK graphics toolkit checks which graphics objects that are associated with the current window and sends the graphics object to the `opengl_renderer` class. Here the graphics objects are sent to a method designed for rendering that specific object. In contrast, the standalone application always draws the same kind of object, so there is much less overhead. We should note that Octave uses immediate mode rendering to draw the axis, whereas the standalone application currently does not show any axis.

In Figure 7.7 to Figure 7.11, we see that the two implementations of the `plot_grid` methods give the highest frame rate of all the tested methods, regardless of modification ratio. From the results it is clear that there is a very large performance benefit from keeping the data on the GPU without transferring new data from the CPU. Both the standalone implementation and the Octave implementation of `plot_grid` allows for easy interaction in terms of rotation and dynamically changing cut planes for the smallest grid, and partly for the grid with $350,000$ cells. For the two largest grids the frame rate drops below 15-20 fps when the modification ratio increases, meaning that interaction with the grid is difficult.

Table 7.4: Results for the `plot_grid` method measured in frames per second.

Method	Ratio	50k cells	350k cells	800k cells	1,600k cells
(Octave)	1:1000	125.0	108.7	90.0	69.4
`plot_grid`	1:100	118.7	77.5	47.5	25.3
	1:10	78.6	29.1	12.4	6.6
	1:2	31.9	7.6	3.1	1.7
	1:1	18.0	4.0	1.6	0.9
(standalone)	1:1000	1478.1	680.0	396.7	191.2
`plot_grid`	1:100	1243.9	364.5	81.5	43.4
	1:10	532.6	80.3	25.6	9.6
	1:2	150.6	18.7	7.3	3.2
	1:1	79.6	9.6	3.7	1.9

7.3.5 Summary of performance tests

The new methods we have developed give a higher frame rate than what we were able to get with standard Octave visualization. In most of the cases, the `patch2` function also has a higher frame rate than Matlab. The `plot_grid` method is faster than both Matlab and original Octave in all of the tests.

When using the MRST functions `plotGrid` and `plotFaces`, the most common work flow will involve a low modification ratio since selecting a subset of the grid is only possible through scripts. With this kind of modification ratio the new methods are more efficient than both original Octave and Matlab in all the test cases. Substituting Octave's original `patch` with the new `patch2` function will change Octave's visualization of grid from being below the threshold of smooth interaction on all of the grids to allow for full interaction with almost all of the grids. It will also make Octave's visualization more efficient than Matlab's plotting routines.

Chapter 8

Conclusions

We will now summarize our findings and give answers to the research questions posed in Chapter 1. In addition we will outline possible further work that can be done on the topic of the thesis.

8.1 Conclusions

We started this thesis by posing two questions:

1. Is it possible develop new methods for visualizing MRST grid properties and simulation results in Octave that makes Octave's visualization functionality equal to that of Matlab?

2. Are we able to develop new methods that support more advanced visualization by using current hardware and modern rendering techniques, and are we able to integrate these methods into Octave?

The foundation for answering the first question was given in Section 6.2 and Chapter 7, where we developed a new `patch` function and tested its functionality and performance. By using the new `patch` we can achieve almost the same functionality for visualizing grid properties and simulation results in Octave as in Matlab. In addition, the new `patch` function offers much higher frame rates than both standard Octave, and Matlab, which makes interaction with the grids much smoother. From our point of view, where the focus has been visualization methods, we conclude that it is possible to make Octave's visualization of MRST grids equal Matlab.

Octave's visualization of MRST grids still has certain shortcomings due to compatibility issues between Octave and Matlab and some missing functionality. We must, however, note that Octave is under constant development, and future releases will probably further decrease the gap in functionality between Matlab and Octave.

The answer to the second question is found in Section 6.3. Here we described the development of a new method for visualizing MRST grids with support for more interactive cut planes which is an example of advanced functionality beyond what Matlab can offer. In addition, to this new functionality, we found in Chapter 7 that it has the highest frame rate of all the methods we have tested. Although we integrated the method itself into Octave, we decided against implementing control of the interactive cut planes through the FLTK graphics toolkit, because of the large changes we would have been forced to do in the Octave source code.

This functionality was implemented in the standalone application, with glut as window toolkit, and communication with Octave through oct-files. This application is not meant as a real alternative to a graphics full graphics toolkit. Instead it is made to illustrate what we can accomplish in terms of functionality and performance when not using the rather big Octave framework. From the performance results in Chapter 7, we see that even though the implementation of `plot_grid` in the standalone application offers the highest frame rate, we are not able to offer full interactivity on larger grids when often changing the cut plane. However, it will be much easier to perform optimizations when working in the more lightweight standalone application, than in the big Octave system.

Another very important argument for not implementing this functionality in Octave, is that it would require large changes in how the data flow in the graphics part of the source code is performed. This would mean that every time a new Octave version is released, a lot of work would have to be done to get the function to work. Alternatively, we would have to stick to the usage of one specific Octave version, which does not make much sense, as a great benefit of using open-source products is their flexibility and frequent releases. This means, that if one was to implement other more advanced visualization methods, they should also be implemented in a separate backend, instead of being integrated directly in the existing Octave source code.

A very central conclusion we can draw is that as much data as possible should be kept on the GPU, and we should minimize the number of times we transfer data from the CPU to the GPU. This will, if implemented correctly, give a large positive impact on performance. When only sending data about vertices and faces, like we do when using MRST routines, we are not able to take full advantage of the power in modern GPUs. By giving the GPU access to the entire description of the grid it is possible, by using modern shader techniques, to offer advanced, high-performance visualization.

The methods we have developed in this thesis are too specialized for MRST to get accepted as part of Octave. Nevertheless, it is possible to present some observations on Octave's rendering code. As it is today, Octave's rendering makes little use of the potential power in the GPU. By reorganizing data flow in the render code, it should be possible for Octave to move away from immediate mode rendering, and instead store geometric data in buffer objects on the GPU. If combined with a smarter usage of the `GLUtesselator`, this would give a

significant improvement in performance.

8.2 Further Work

A master thesis is limited in time and scope, and there are several topics that would have been interested to investigate further. Examples of such topics are:

Parallelize the Ear Clipping Algorithm. The task of triangulating polygons with the Ear Clipping Algorithm is embarrassingly parallel, as there is no dependency between the triangulation of separate polygons. Since most modern CPUs are multi-core, it should be possible get a significant speed up by utilizing CPU-threads when triangulating polygons.

More focus on memory efficiency. The industry is very concerned with visualizing large grids, from tens of millions cells up to a billion cells and more. To make this possible we would need to find more memory efficient methods for representing the grids. In addition it would be necessary to develop more sophisticated memory monitoring mechanisms so that grids are rendered correctly even though the entire grid does not fit in GPU memory.

Optimize the `plot_grid` function. Even though the `plot_grid` function offers the highest frame rate of all the methods we have tested, it would be beneficial for cut plane interaction to further improve its performance. An option for improving performance could be to not always use transform feedback. When constantly changing cut plane, the use of transform feedback will hinder performance. The whole point of using transform feedback is then we only have to use the heavy geometry shader when necessary, and most of the time we can make use with light-weight shaders. However, when we constantly change the cut plane we need to use the geometry shader each time. Therefore, there is nothing to gain by first rendering to a buffer with the complex geometry shader, and then using light-weight shaders to render from that buffer.

Another possible improvement could be to have the vertex shader that checks which cells that are part of the subset process more than just one cell. One option could be to process 32 cells together, and use a single unsigned integer (32 bits) to store the results, instead of a whole integer per cell. This would make the method more memory efficient.

More advanced visualization. By implementing interactive cut planes, we have introduced more advanced visualization of MRST grids in Octave. However, there are several other types of visualization that would be useful to implement. Examples are volume visualization and isosurfaces.

Appendix A

Obtaining Source Code

This appendix describes how the source code developed as part of the thesis can be obtained. The requirements for using the software are also listed.

Getting the code. The source code that has been integrated into Octave is available as a git[1] patch against Octave version 3.6.1. To read (and install) the source code you will therefore need git installed on your system, and a copy of Octave 3.6.1. Instructions for how to download and apply the patch are available at the following address:

http://folk.uio.no/larsjr/master_thesis

After the git patch has been applied, the source code for the new visualization methods are found in the folder `src`. The code is spread around in different files, but the most central methods (`draw_patch` and `draw_unctructured_grid`)are contained in the files gl-render.cc and gl-render.h.

The code for the standalone application is also available from the same address in the archive

`standalone.tar.gz`

This archive contains files for triangulating and rendering grids, together with an oct-interface for communication with Octave.

System requirements. The code has been developed for Unix/Linux systems, and tested on Ubuntu 10.04 and Ubuntu 11.10. It requires OpenGL version 4.2 or newer. To install and run the software you will need the following libraries

[1] http://git-scm.com/

- glm[2]
- freeglut [3]
- GLEW [4]

[2]http://glm.g-truc.net/
[3]http://freeglut.sourceforge.net/
[4]http://glew.sourceforge.net/

Appendix B

Octave function to extract a subset of a MRST grid

This appendix contains the source code for the Octave/Matlab functions to find a subset of a grid, based on a plane equation that we use when performing the tests in Chapter 7. The functions are fully vectorized to achieve high performance. The first function returns the global face indices of the subset, whereas the second returns the global cell indices. Both functions accept a grid structure and a plane equation, represented as a 1×4 vector.

```
function face_selection = face_halfplane_selection(G, ...
                            plane_equation)

    % Find the nodes that are on the positive side of the half
    % plane. If node i is part of the subset draw_nodes(i) == 1,
    % if not nodes_in_subset(i) == 0.
    nodes_in_subset = ([G.nodes.coords ones(G.nodes.num, 1)]...
                        *plane_equation' >= 0);

    % Reconstruct the first column in G.faces.nodes that
    % describe which global face number a node belongs to.
    face_indices = rldecode(1 : G.faces.num, ...
                    diff(G.faces.nodePos), 2) .';

    % Find the faces that has at least one node on the positive
    % side of the half plane. If faces_in_subset(i) > 0
    % then global face i is part of the subset. If
    % faces_in_subset(i) == 0 global face i is not part
    % of the subset.
    faces_in_subset = accumarray(face_indices, ...
                        nodes_in_subset(G.faces.nodes));
```

```
    % Find the indices of the faces that are part of the subset.
    face_selection = find(faces_in_subset);
end
```

```
  function cell_selection = halfplane_selection(G, ...
                                plane_equation)

    % Find the nodes that are on the positive side of the half
    % plane. If node i is part of the subset draw_nodes(i) == 1,
    % if not nodes_in_subset(i) == 0.
    nodes_in_subset = ([G.nodes.coords ones(G.nodes.num, 1)]...
                         *plane_equation' >= 0);

    % Reconstruct the first column in G.faces.nodes that
    % describe which global face number a node belongs to.
    face_indices = rldecode(1 : G.faces.num, ...
                    diff(G.faces.nodePos), 2) .';

    % Find the faces that has at least one node on the positive
    % side of the half plane. If faces_in_subset(i) > 0
    % then global face i is part of the subset. If
    % faces_in_subset(i) == 0 global face i is not part
    % of the subset.
    faces_in_subset = accumarray(face_indices, ...
                         nodes_in_subset(G.faces.nodes));

    % Reconstruct the first column in G.cells.faces that
    % describe which global cell number a face belongs to.
    cell_indices = rldecode(1 : G.cells.num, ...
                    diff(G.cells.facePos), 2) .';

    % Find the cells that has at least one face on the
    % positive side of the half plane. If
    % cells_in_selection(i) > 0 then global cell i is
    %part of the subset. If cells_in_selection(i) == 0
    % global cell i is not part of the subset.
    cells_in_selection = accumarray(cell_indices, ...
                           faces_in_subset(G.cells.faces(:,1)));

    % Find the indices of the cells that are part
    % of the subset.
    cell_selection = find(cells_in_selection);
end
```

Bibliography

[1] J. E. Aarnes, T. Gimse, and K.-A. Lie. "Geometrical Modeling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF". In: ed. by G. Hasle, K.-A. Lie, and E. Quak. Springer, 2007. Chap. An introduction to the numerics of flow in porous media using Matlab.

[2] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering*. Third. A K Peters Ltd, 2008.

[3] S. Baker. *freeglut*. URL: http://freeglut.sourceforge.net/ (visited on 04/24/2012).

[4] I. P. on Climate Change. *Carbon Dioxide Capture and Storage: Special Report of the Intergovernmental Panel on Climate Change*. Tech. rep. 2005.

[5] C. Dyken. "Personal communication". Visualization of Corner-Point Grids.

[6] J. W. Eaton, D. Bateman, and S. Hauberg. *GNU Octave Manual Version 3*. Network Theory Limited, 2008.

[7] J. W. Eaton. *Homepage of the Octave Project*. URL: http://www.gnu.org/software/octave/ (visited on 04/24/2012).

[8] *FLTK*. URL: http://www.fltk.org/ (visited on 04/23/2012).

[9] T. F. S. Foundation. *GNU Autoconf*. URL: http://www.gnu.org/software/autoconf/ (visited on 04/24/2012).

[10] T. F. S. Foundation. *GNU Automake*. URL: http://www.gnu.org/software/automake/automake.html (visited on 04/24/2012).

[11] *gnuplot*. URL: http://www.gnuplot.info/ (visited on 04/23/2012).

[12] M. Hall. "The future is open for business: open source tools for the geoscientist". In: *first break* 28 (6 June 2010), pp. 119–123.

[13] R. S. W. Jr et al. *OpenGL SuperBible*. Addison-Wesley, 2011.

[14] J. Kessenich, D. Baldwin, and R. Rost. *The OpenGL Shading Language*. The Khronos Group Inc. Sept. 2011.

[15] K.-A. Lie. "Reservoir simulation and MRST". Book manuscript.

[16] E. Lindholm et al. "Graphic and Computing Architecture". In: *IEEE MICRO*. Vol. 28. IEEE Computer Society, 2008, pp. 39–55.

[17] D. Luebke, E. d'Eon, and E. Enderton. "GPU Architecture: Implications and Trends". In: *GPU Gems 3*. Siggraph2008. NVIDIA Research, 2008.

[18] Nvidia. *NVIDIA GeForce GTX 580 Specifications*. URL: http://uk.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications (visited on 04/24/2012).

[19] OpenGL. *History of OpenGL*. URL: http://www.opengl.org/wiki/History_of_OpenGL#OpenGL_4.0_.282010.29 (visited on 04/24/2012).

[20] *OpenGL*. URL: www.opengl.org (visited on 04/22/2012).

[21] J. O'Rourke. *Computational Geometry in C*. Second. Cambridge, 1998.

[22] D. K. Pointing. "Corner point geometry in reservoir simulation". In: *Proceedings of the 1st European conference on Mathematics of Oil Recovery, Cambridge*. Clarendon Press, 1989, pp. 45–65.

[23] P. Read and M.-P. Meyer. *Restoration of motion picture film*. Butterwroth-Heinemann, 2000.

[24] R. J. Rost and B. Licea-Kane. *OpenGL Shading Language*. Third. Addison-Wesley, 2010.

[25] J. Sanders and E. Kandrot. *CUDA by Example*. Addison Wesley, 2011.

[26] M. Segal and K. Akeley. *The OpenGL Graphics System: A specification ,Version 4.2 (Compatiblity Profile)*. The Khronos Groupc Inc. Aug. 2011.

[27] D. Shreiner et al. *OpenGL Programming Guide*. Sixth. Addison Wesley, 2008.

[28] SINTEF. *Homepage of the MRST project*. URL: http://www.sintef.no/Projectweb/MRST (visited on 04/23/2012).

[29] *The Mesa 3D Graphics Library*. URL: www.mesa3d.org (visited on 04/25/2012).

[30] D. Wolff. *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing, 2011.